Francisco Miguel Carmo Godinho

Bachelor of Computer Science and Engineering

Bringing Order into Things

Decentralized and Scalable Ledgering for the Internet-of-Things

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Henrique João Lopes Domingos, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Matthias Knorr
Raporteur:	Fernando Manuel Valente Ramos
Member:	Henrique João Lopes Domingos

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Bringing Order into Things - Decentralized and Scalable Ledgering for the Internet-of-Things

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my grandfather.

Acknowledgements

In the first place, I would like to thank my thesis advisor, Prof. Henrique João Domingos, for the opportunity he gave me to work with him. This thesis was only possible thanks to his constant support and enthusiasm. I also want to thank the Faculty of Sciences and Technology of the NOVA University of Lisbon and its incredibly talented professors that have taught me everything I know over these years. I will miss calling this university home.

A word of appreciation goes to my parents and grandparents for their never-ending support and encouragement to finish my degree. To my partner in life for the incredible motivation she has given me. To my brother, sister-in-law and nephews who are far away from sight but near my heart. To Benji. And last, but not least, to my friends and colleagues without whom I would never be who I am today. I shall cherish every opportunity of personal growth I was given during my studies along with every dear friend I made along these years.

Abstract

The Internet-of-Things (IoT) is simultaneously the largest and the fastest growing distributed system known to date. With the expectation of 50 billion of devices coming online by 2020, far surpassing the size of the human population, problems related to scale, trustability and security are anticipated. Current IoT architectures are inherently flawed as they are centralized on the cloud and explore fragile trust-based relationships over a plethora of loosely integrated devices, leading to IoT platforms being non-robust for every party involved and unable to scale properly in the near future. The need for a new architecture that addresses these concerns is urgent as the IoT is progressively more ubiquitous, pervasive and demanding regarding the integration of devices and processing of data increasingly susceptible to reliability and security issues.

In this thesis, we propose a decentralized ledgering solution for the IoT, leveraging a recent concept: blockchains. Rather than replacing the cloud, our solution presents a scalable and fault-tolerant middleware for recording transactions between peers, under verifiable and decentralized trustability assumptions and authentication guarantees for IoT devices, cloud services and users. Following on the emergent trend in modern IoT architectures, we leverage smart hubs as blockchain gateways, aggregating, pre-processing and forwarding small amounts of data and transactions in proximity conditions, that will be verified and processed as transactions in the blockchain. The proposed middleware acts as a secure ledger and establishes private channels between peers, requiring transactions in the blockchain to be signed using threshold signature schemes and group-oriented verification properties. The approach improves the decentralization and robustness characteristics under Byzantine fault-tolerance settings, while preserving the blockchain distributed nature.

Keywords: Blockchains, Internet-of-Things (IoT), Decentralized Ledgering, Decentralized Trust, Threshold Signature Schemes, Scalable Fault-Tolerant Ledgering Middleware

Resumo

A Internet-das-Coisas (IoT) é o sistema distribuído maior e de mais rápido crescimento conhecido à data de hoje. Espera-se que 50 mil milhões de dispositivos se ativem e interliguem até 2020, o que ultrapassará largamente o tamanho da população humana e colocará problemas complexos de escala, confiabilidade e segurança. Verifica-se que as atuais arquiteturas de IoT têm limitações inerentes, devido à centralização na *cloud* e à integração pouco regulada de uma panóplia de dispositivos com base em relações de confiança frágeis para os participantes envolvidos, resultando na incapacidade das plataformas de IoT de escalar eficientemente no futuro próximo. A necessidade de novas arquiteturas que abordem estas questões é urgente, à medida que a IoT se torna cada vez mais ubíqua, disseminada e passe a exigir a integração de dispositivos e processamento de informação cada vez mais sensíveis do ponto de vista da fiabilidade e segurança.

Nesta tese, propomos uma solução de registo de transações descentralizado para a IoT, beneficiando de um conceito recente: *blockchains*. Em vez de substituir a *cloud*, a nossa solução apresenta-se como um *middleware* escalável e tolerante a falhas, capaz de registar transações entre participantes com garantias de autenticação entre dispositivos de IoT, serviços de *cloud* e os próprios utilizadores, de forma mutuamente auditável e sem dependência de bases de confiança externas. Seguindo ainda a orientação recente nas novas arquiteturas para a IoT, a nossa abordagem utiliza *smart hubs* como portais de comunicação com a *blockchain* que permitem a agregação de tráfego e o processamento de pequenos conjuntos de dados em condições de proximidade, que serão verificados e processados como transações na *blockchain*. O *middleware* perspectivado age como um registo de transações seguro e confiável e estabelece canais privados entre participantes, providenciando assinaturas de transações com base em esquemas de assinatura de limiar, com verificação e auditabilidade orientada a grupos de participantes, para um aumento de descentralização e robustez em conjugação com requisitos de tolerância a falhas bizantinas, preservando a natureza distribuída da tecnologia *blockchain*.

Palavras-chave: *Blockchains*, Internet-das-Coisas (IoT), Registo de Transações Descentralizado, Esquemas Criptográficos de Assinaturas de Limiar, *Middleware* de Registo de Transações Escalável e Tolerante a Falhas

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context and Motivation

Blockchain technology first emerged in 2008 as a proposition for a cryptocurrency widely known as Bitcoin, allowing for secure peer-to-peer transactions without relying on third-party entities, such as financial institutions and banks [54]. Considered by many as a disruptive force on both the industry and the academia, and viewed by others as somewhat hyped and yet to mature, blockchains are a novel concept that's growing beyond digital currencies, steadily becoming a foundational technology adopted in other areas due to its immense potential for transforming the traditional industry by providing decentralization, security, persistency and auditability properties [45, 62, 79].

One of the areas where blockchain's properties seem most beneficial in its potential is the approach of a new generation of architectures for the Internet-of-Things (IoT). The IoT is a new paradigm shift that promises to seriously impact our everyday lives with the presence of intelligent and ubiquitous systems that will interact between themselves and directly with us humans, whether or not we are aware of it. Expected to grow at an extraordinary pace, reaching around 50 billion devices by 2020 [39], the IoT already needs an urgent reboot [55], mainly because of high infrastructure costs due to the usage of centralized platforms, proprietary server farms and cloud services, the reliance on trust-based models, and the overall lack of privacy and security concerns. As the IoT market keeps growing, the explosion of data to be transmitted to the cloud in current centralized models will lead to cloud providers struggling with their capacity and resources. The impetus for this reboot is further strengthened by today's IoT platforms offering no possibility for scrutiny or control by end users in terms of privacy, security or trustability conditions. No two devices can interact directly without communication passing through service providers, while commercially available IoT platforms and services maintain their

own data silos for personalized service and profiling, without users being able to control how their data is used and logged and with the possibility of that same data being sold to third-parties or mishandled by providers.

In fact, it seems that the blockchain, an invention once fueled by the need of replacing the increasingly prevalent trust-based model in electronic payment systems with a more secure and robust model tolerant to fraud attempts and to third-party manipulation could extend its use-case into the IoT and solve some of its growing entropy. For such synergy to occur, however, some challenging obstacles that arise from both worlds need to be studied and tackled first. Blockchains currently suffer from two considerable problems: scalability and privacy, with repercussions of such concerns in achieving the best balance in performance (transaction throughput and latency conditions), as well as in membership management of the participants involved and in the reliability guarantees under Byzantine-fault tolerance assumptions. Scalability has been a continuous source of debate on how to improve transaction processing throughput and latency without compromising system security and decentralization [20, 74]. Privacy has seen ongoing efforts on how to deal with data privacy and traceability concerns, since transactions are globally published across peers [5, 78]. On the other hand, the IoT is also facing major challenges [60], such as the heterogeneity of devices properties and the proliferation of proprietary solutions as specific IoT ecosystems, the lack of standards for interoperability and for communicating parties, regulated protocols and technologies, and the diversity of support for different cryptographic primitives and algorithms.

Recent edge computing models have also emerged to mitigate the privacy issues in the growing market of the IoT by providing local processing and storage capabilities to the edge, leveraging smart hubs that aggregate, filter, process and control data-flows, intermediate different protocols by protocol translation, and orchestrate sets of IoT devices in local-controlled environments, avoiding the need of direct communication with cloud-provider services and applications [10]. These hubs allow for better privacy controls by selective filtering and obfuscating user data while aggregating and pre-processing data on the edge, also avoiding huge amounts of sensitive data to be sent up to the cloud. A vision of merging these new architectural models for the IoT with blockchain-enabled data repositories and logging, regarded as *blockchained* IoT platforms, seems to be an interesting direction to solve some of the scalability, security, privacy and trustability issues of current IoT platforms and applications.

1.2 Objective and Expected Contributions

Problem Statement. Current security models for the IoT are centralized by nature or cloud-based intermediated under no control by end users, relying on a central authority for each specific service provider to orchestrate device communication, data storage and logging of operations between several nodes. While this approach is acceptable assuming an honest central authority and a modest number of devices, a preferable solution would

be one that does not rely on centralized models of trust in service providers or in third-party interventions of entities assumed as trusted services. The decentralization of such functions under independent auditing guarantees is crucial, given the exponential growth of the number of IoT devices and the need for a model that is able to scale accordingly and provide reliability without a single point-of-failure or being too costly to maintain. Therefore, the question that the dissertation addressed is e following:

How can we address the foundations, services and mechanisms, to design new and more reliable IoT platform architectures, as to improve trustability, scrutiny and scalability guarantees, taking advantage of blockchain-enabled logging and decentralized ledgering properties?

Objective. Our hypothesis for the research question in the problem statement is that it is possible to improve the trustability properties of IoT platforms by introducing mechanisms to perform information flow control via local smart hubs, providing gateway facilities supported by blockchain-enabled decentralized data management conditions. We believe that the approach can address better conditions to scale and to control the trustability assumptions of the IoT operation. For this purpose, we must research on how to efficiently reshape the common centralized security model of the current cloud-enabled IoT platforms, into a decentralized approach, reducing the need for centralized trust authorities, and simultaneously considering the possible integration of the diversity of IoT devices, that can range from objects as mundane as toasters and doorknobs to highly complex technology as rentable cars and industrial equipment, each with their own different computational capabilities and resource limitations. Figure 1.1 reflects a high-level illustration of our intent and on how we wish to harness these concepts to transition from traditionally centralized IoT architectures to a new generation of decentralized and independently auditable architectures.

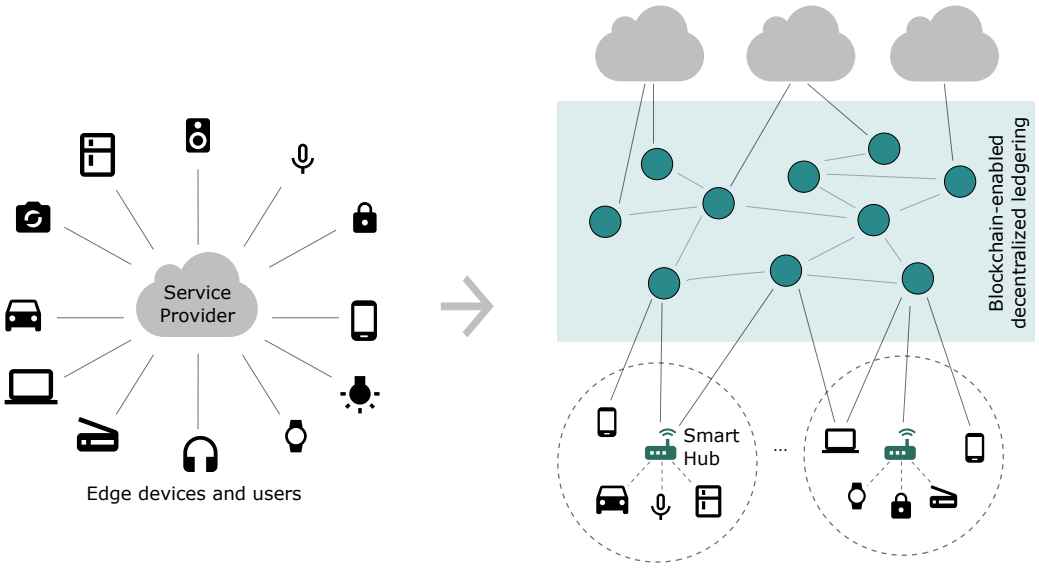

Figure 1.1: High-level system model proposal

To support our thesis statement, we must introduce improved mechanisms in the blockchain for a better management of decentralized trust in scalability conditions. We can see this approach as a clean slate approach for a new generation of open IoT platforms, in which the users have full control over their devices, autonomy to audit trustability conditions and a better control of the sensor data collected in the edge of the IoT. In this approach, we will add to the notion of the smart hub edge-based architectures and provide middleware services with the ability for decentralized ledgering and group-oriented verifiable signatures and verifiable transaction logging, allowing for a global autonomous validation of transactions involving data sharing between devices, in a fully decentralized data management model enabled by a base blockchain platform.

Research Contributions. In concordance with the objective defined above, we emphasize the following contributions as a set of relevant outcomes from the thesis elaboration:

- An analysis on effectiveness, properties and support provided by different blockchain platforms, surveying their provided services and mechanisms, in order to be addressed as candidates for leveraging extensibility requirements in enhancing decentralized ledgering and reliability foundations for blockchain-enabled IoT platforms;
- In the previous contribution we include a detailed analysis of different blockchain platforms, studying each platform in terms of different characteristics, including: system architecture, software engineering approaches, reliability and scalability conditions, as well as, programming support, which in turn, includes the analysis of support for smart-contracts regulating supported transactions, regarding expressiveness conditions, openness and extensibility possibilities;
- A system model capable of acting as a decentralized ledger for the IoT that is able to deviate the blockchain scalability concerns away from the IoT and provide a robust cryptographically-verifiable middleware layer and an autonomous verification of trustability conditions that protects every party involved;
- An innovative transaction and smart contract verification protocol that resorts to threshold signatures to further decentralize blockchain architecture, capable of outperforming multi-signature schemes under Byzantine fault-tolerance guarantees and requiring smaller ledger block sizes;
- A powerful extension for smart contract specification that allows contracts not only to specify the properties of an application running on a blockchain system but also the properties of the system itself and on how transaction flows should occur;
- A fully functional prototype implementation based on our system model and its respective experimental assessment, where smart hubs can be leveraged to provide scalable gateways to blockchain-enabled IoT platforms with increasing stress conditions and resort to lightweight communication protocols acceptable for IoT resource-constricted devices.

1.3 Document Organization

Following this first chapter, the remainder of this document is organized as follows:

- **Chapter** 2 focuses on a series of relevant related works in a top-down fashion, starting from an understanding of current IoT cloud-based platforms and the inherent centralization problems related to these architectures, to how blockchains work and an analysis of existing blockchain platforms, all the way down to threshold cryptography and its applicability to the blockchain transaction verification flows;
- **Chapter** 3 describes our system model and architecture for a decentralized and scalable ledgering platform for the IoT, inspired by the related work;
- **Chapter** 4 outlines our prototype implementation effort for a system inspired by the system model in the previous chapter;
- **Chapter** 5 presents the results of our experimental assessment over the implemented prototype and our critical analysis of each benchmark;
- **Chapter** 6 wraps up this document with a set of concluding remarks on the conducted work, stating left open issues and future work.

Chapter 2

Related Work

In this chapter we address the existent related work of our thesis. We first present a background of the study to familiarize the reader with each topic we describe ahead. We start the related work by addressing the Internet-of-Things (IoT), cloud-first IoT architectures and their handicaps. We then introduce blockchains and their properties, proceeding to evaluate the state-of-the art of blockchain platforms. Finally, we present threshold cryptography and theorize on the possibility of leveraging threshold signature schemes for blockchain transactions to increase decentralized trustability properties of typical blockchain architectures. We finish the chapter with a critical analysis of these subjects.

2.1 Background

The inability of current cloud-first IoT architectures to provide independent auditability properties to every party involved with decentralized trustability assumptions is increasingly critical with each new IoT device present in our lives. Current systems, which are heavily dependent upon centralized cloud infrastructures, lack in robustness and privacy concerns, and are significantly susceptible to faults and fraudulent behaviour from service providers. Concerns that cloud providers will not scale in the foreseeable future as the IoT grows at an astounding rhythm and that the fierce competition between IoT platform manufacturers will drive to the increase of closed IoT environments and vendor lock-in practices are also paramount. Thus, an alternative architecture that specifically addresses the issues of robustness, centralization and scalability is needed.

The fact that this new architecture may be given by a technology that we popularly acknowledge as being the backbone of cryptocurrencies as Bitcoin and Ethereum may be unexpected at first. However, blockchains have clear practical beneficial properties that can be applied to various scenarios, as they provide a truly decentralized approach to

modern distributed systems and data-repository services, that has yet to be replicated by other systems. Foremost, they allow for cryptographically secure and verifiable ledgering. As such, various platforms that harness blockchain technology have emerged over the years [5, 12, 28, 33, 40, 56], enabling for innovative decentralized applications that are better prepared to answer the scalability and dependability requirements imposed by modern systems.

For the sake of simplicity, a blockchain can be viewed as a public ledger in which multiple peers register and verify transactions between themselves. These transactions are recorded in the form of blocks, which are then chained together forming the actual blockchain, acting as a permanent database for the transaction records [54]. The blockchain is not bound to a central authority and is instead distributed across the multiple peers that compose the network, hence the property of decentralization. To add new blocks to the chain, nodes have to *mine* the most recent block, which requires solving a computational puzzle based on cryptographic hash functions, and present a proof-of-work [54] to other peers so that they can continue on producing the next block.

We will approach some of the state-of-the-art of existent blockchain platforms in bigger detail in the following sections and study different points-of-view on how to implement the blockchain model and relevant extensions. Perhaps the most powerful evolution to date to be implemented on the blockchain is the concept of smart contracts [70] – special scripts that reside on the blockchain providing it the ability of enforcing and cryptographically verifying distributed workflows – which is a source of immense interest for the IoT domain since it enables the automation of complex multi-step processes [16].

Before the blockchain is successfully merged into the IoT ecosystem, a larger comprehension of what cryptographic primitives and multiparty computing protocols to use is needed in comparison with traditional environments that are not as diverse and resource-limited in nature. Promising advances have already been made, with lightweight encryption standards and ECC (Elliptic Curve Cryptography) being pushed forward as a possible future for resource-constrained devices [60], as well as the rise of new architectures that are able to reduce some of the overhead of the blockchain protocol by outsourcing the protocol to a back-end network more capable of handling its overhead [24].

2.2 The Internet-of-Things (IoT)

The Internet-of-Things (IoT for short) can be defined as the collection of everyday devices and objects that are embedded with electronics allowing them to compute and communicate via some type of network. This definition covers a lot of physical objects we know today that are already interconnecting with each other through virtual components practically everywhere by leveraging micro-controllers, network adapters, sensors and actuators. Such devices are progressively more aware of their surroundings and more capable of capturing, producing and reacting to information for various purposes (e.g. motion detection, speech recognition, surveillance and intrusion detection) [60]. This

ubiquity has lead the IoT far away from being a concept, and it has been growing and evolving at a stunning speed, expecting to hit the 50 billion device mark by 2020 according to current estimates [39]. Its sheer potential opens up innovative and increasingly complex concepts such as smart homes, domotic-oriented office automation, smart energy grids, and smart cities by establishing networks where several specialized devices can collaborate to provide a number of pervasive services [4, 24].

2.2.1 IoT Platforms and Architectures

Today's most well known use-case of IoT systems are at the smart home setting. The need and usefulness for automating and remotely controlling several devices in the common household by connecting them over the Internet inspired the idea of integrating microcontrollers into almost everything. However, as manufacturers produced more and more smart devices over time, using cost-effective hardware components for specialized functions, the IoT got increasingly heterogeneous, composed by multiple devices with very distinct capabilities, most of which considerably resource-constrained.

To address this, current IoT platforms generally apply a cloud-first architecture, meaning they offload resource-intensive tasks to cloud services, since it is infeasible to perform complex application logic on the severely constrained hardware of most devices. As such, IoT platforms usually come in the form of proprietary cloud gateways which the user can control via some sort of desktop or smartphone application. The cloud services, in turn, orchestrate the IoT devices connected to it according to the user's commands and to deployed internal application logic. Data generically flows back and forth between the cloud and the edge, with the latter collecting environment information and actuating upon it and upon the cloud service's commands. In most cases, users have no direct control of what information is sent/received to/from the cloud besides what the service provider intends to give them control of. Still, the motivation behind cloud-based approaches is clear: they allow for an unique centralized point of management, orchestration and monitoring of IoT environments, which have the potential for reaching overwhelming sizes of thousands, millions and perhaps even billions of devices that are impossible to address simultaneously otherwise. At the same time, they deviate resource-intensive tasks from the IoT. And even though newer IoT devices may be less constrained in the future, older devices or extremely specialized types of devices (e.g. door locks, light bulbs, thermometers) that already compose a big part of the IoT are unlikely to be updated [17].

In recent years, smart home architectures started introducing a new intermediary device at the edge level, called a smart hub, in an attempt to unify the plethora of different devices, independently of manufacturer or proprietary software, under a single unified control interface and a standardized set of communication protocols (e.g. Apple HomeKit, Amazon Echo, Samsung SmartThings [2, 3, 61]). These architectures closely resemble the depiction on Figure 2.1. These centralized hubs have the benefits of performing data aggregation, storage and processing on the edge and communicate with cloud services

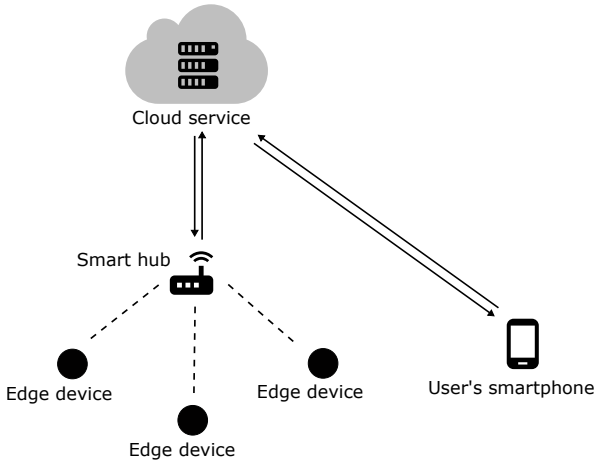

Figure 2.1: Depiction of cloud-based IoT architectures

on behalf of the smart devices only when needed, improving overall user experience and minimizing latency for tasks that do not necessarily need the raw power of the cloud.

2.2.2 IoT Centralization Concerns: Privacy, Scalability and Robustness

Cloud-based platforms have the ability of rearranging the entropy of IoT networks into simple vertical topologies with a centralized point of control, data storage and processing. While practical benefits are evident, these vertical architectures raise some concerns with potentially negative and harmful consequences, particularly regarding to the privacy of user's personal data, the scalability of modern cloud systems and the robustness of IoT architectures. These three problems all originate from the *centralization* of the whole system to a single point-of-failure and trust: the cloud. Our thesis is particularly concerned in finding solutions for the latter two issues.

Privacy. As mentioned before, users have little control over what information do their devices send to the cloud. During the normal lifecycle of edge devices, cloud systems are used to store and process all needed data for posterior use by the whole system. Simultaneously, cloud providers collect information for analytic and statistical purposes in order to provide a better quality of service. This continuous process of collecting data in the cloud requires the user to trust the service that his personal and sensitive information is kept confidential and secured, and that the service is not going to attempt to use it in a fraudulent manner. This is a specially strong assumption, as seen from recent real cases of theft and unauthorized disclosure of personal information [26, 31, 38, 52], proving that this trust relationship has been broken a worrisome number of times, with data being used for illegitimate purposes.

Scalability. As the number of connected IoT devices grows at an exponential rate, so does the amount of data they generate, directly increasing the workloads that cloud services (and consequentially data centers) need to handle. Current predictions estimate that these workloads will increase massively, as far as 750% up until 2019 [76]. Given

that current IoT architectures rely on the cloud extensively these estimates serve as a dire warning that the consumption of resources by the IoT will far exceed the capability of the cloud to provide them. Current systems will not be able to scale to the needs of the IoT, and a redesign of existing architectures is therefore urgently needed. Some authors have proposed to address the scalability problem at the edge with the integration of edge computing models such as the one in [10] as part of the architecture of IoT platforms. These models rely on the deployment of edge nodes with reasonable hardware and capable of storing and processing moderate amounts of data close to the resource-constrained IoT devices to perform local data aggregation, storage and computing tasks, minimizing the volume of data and traffic sent to the cloud, resorting to it only when further processing is needed. This model seems to be in line with the emergent smart hub technology described previously and allows for a better privacy control, since data can be processed locally and the hub may filter subsets of that same data to be sent to the cloud for analytics or backup.

Robustness. Current IoT models do not offer satisfactory security measures for all of the parties involved, depending too much on centralized entities that are susceptible to fraudulent behaviour. A network where every device is subjugated to a central authority is a network that is ultimately controlled by a single point-of-failure. More importantly, there is no direct form for users and devices to verify and audit if certain workflows are valid or authorized without depending on the central authority to do so [4, 16]. The problem with centralization is even more evident in critical systems like financial and banking operations, healthcare systems and military networks, in which people's assets and lives may be compromised by faulty systems [4]. A security-through-transparency *democratic* decentralized and trustless ledgering solution, capable of securely distributing information in cryptographically verifiable workflows across IoT devices is therefore needed, and can be leveraged through the use of emergent blockchain technology according to several authors [4, 16, 55, 64]. For instance, in [24] the authors propose a decentralized overlay network using blockchains that uses the cloud merely for storing data, although it assumes the cloud as part of its trust computing base. In this model, smart hubs connect to representative nodes in the overlay network which is responsible for providing robustness and auditability while deviating the blockchain protocol from edge devices. In [64], despite the absence of smart hubs in their architecture, the authors propose the use of a blockchain layer to mediate access control and disrupt the need of trust in cloud systems in a similar way to [24].

2.3 Blockchains

In this section we present blockchain technology and relevant concepts. For the context of this thesis our focus is on blockchain technology itself and not on Bitcoin or any other type of cryptocurrency, even though we might resort to such systems for illustrative purposes.

2.3.1 Relevant Concepts in Blockchain Technology

A fundamental problem of distributed systems is consensus, which requires different processes, or participants in the same network, to decide and agree on a given data value in order for the system to maintain overall consistency. The consensus problem has been around since the birth of distributed systems and has been formally described in the literature [18]. Consensus mechanisms have to be properly designed if they are to be used in real world applications. An ordinary aspect of any system is that it can fail, either by crashes, message omissions or Byzantine behaviour [50] (faults caused by software/hardware errors or malicious attacks on nodes). Consensus algorithms have to take this fact into account and provide a way to maintain consistency in the presence of a given number of faults. However, in asynchronous networks, this concept collides with a problem known as the FLP impossibility [27], which states that it is impossible to deterministically establish consensus in an asynchronous system where at least one process is able to fail. Nevertheless, various algorithms, such as Paxos and Google's Chubby, are used today in asynchronous settings and a small probability of not reaching consensus is tolerated.

The blockchain, as the name suggests, is a chain, or a list of records, comprised of a continuously growing number of data blocks, linked together in a way that allows participating nodes in a peer-to-peer network to establish a sequential history of transactions[1] and agree on the order that they occurred on the system. To achieve this, blockchains resort to their own consensus mechanism based on cryptographic proof. Blockchains can be thought of as decentralized databases, providing persistency of data, a high level of fault-tolerance, and security, all without the need of trust in a third-party. Decentralization is a key property, since there is no single point of failure in the system, making truly decentralized blockchains resilient to even Denial of Service (DoS) attacks.

2.3.2 Blockchain Operation

Nodes that wish to start a new transaction in the blockchain broadcast it over the network, letting other nodes collect this transaction into a data block – the main unit of the blockchain which is able to hold several different transactions [54]. Every data block in the blockchain contains an *hash* of the previous block, a *nounce* and relevant transaction data, as illustrated by Figure 2.2.

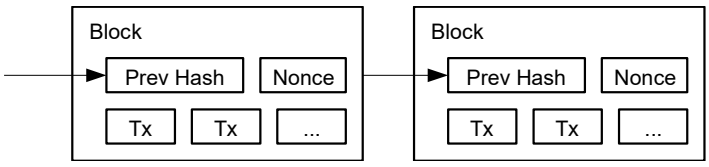

Figure 2.2: Blockchain block structure (from [54])

[1] We use the term *transaction* since that's the terminology used in the literature, but it should be considered in the sense of a generic exchange of information between processes rather than its economic denotation.

The *hash* of the previous block is generated by applying a secure hash function to the content of the block. At the start of the chain lies the genesis block, whose purpose is to act as the beginning of the chain and authenticate the entire chain (backtracking on any block should always yield the same genesis block). The *nounce* is an arbitrary number whose value is determined in a way that ensures that the secure hash function output of the block starts with a sequence of leading zeros. To compute the value of the *nounce*, nodes go through a process known as *mining*, looping through different values for the *nounce* and submitting them to the secure hash function until a result with leading zeros is found, expending CPU power and resources. The number of leading zeros determines the difficulty involved in finding the correct *hash*, with a lesser number of zeroes requiring less computational effort than a bigger number, and is decided by the participants of the network if blocks are being produced at a faster than normal rate. This cryptographic puzzle solving process lies at the heart of blockchain's consensus mechanism, since nodes are required to present the block and the correct *nounce* as a proof-of-work to other participants in the network to be granted the right to add the new block to the existing chain. Upon receival of the proof-of-work, other nodes can quickly verify that it is in fact the correct solution by submitting it to the same secure hash function, and checking if the contained transactions are valid. If both verifications succeed, nodes accept the proof-of-work and begin working on the next block, thus expanding the chain. When two or more nodes solve the mining problem concurrently or a misbehaving node intentionally duplicates blocks of transactions (an attack known as double-spending), a *fork* occurs, splitting the blockchain into different branches. When this happens, miners can work on any of the branches. Eventually, one of them will have more work invested into it and grow in length, so nodes will redirect their effort to it since it will be the longest chain. Due to the possibility of such race conditions, transactions are only confirmed after the block they are contained hits a certain threshold of successors – the bigger number of appended blocks, the slimmer is the probability of existing a concurrent branch.

Mining also gives the blockchain its inherent resistance to tampering and data modification, since to change an existing block an attacker would have to regenerate all the other blocks that come after it, recomputing the proof-of-work for each one. If the network is dominated by an honest majority ($\geq 51\%$), it is impossible that an attacker is able to forge a fraudulent fork that surpasses the length of the honest chain even if spawning multiple nodes, since honest nodes will always mine on the longest chain. Therefore, transactions stored in the blockchain are considered to be irreversible, since attackers would need to subvert the network to be able to forge a chain that reverts them.

The last component of the block is the actual data, which is stored in a Merkle tree structure [53]. The purpose of this structure is that it serves as cryptographic proof of the transactions stored in a block by hashing upward to the tree's root: if transaction data (a leaf node) is tampered with or any node in the tree is rearranged, the resulting root hash value of the tree won't match the original one, and can be detected as invalid. Pruning

can be employed to save disk space without damaging the tree's intended function. Transactions themselves are stored in the form of *coins*, a term originating from cryptocurrency, which are structured objects generally containing the address of the recipient, the data of the transaction and the sender's digital signature proving the authenticity of the coin.

2.3.3 Smart Contracts

While not part of the original architecture of the blockchain, the concept of smart contracts [70] emerged as a way of enabling content-agnostic blockchain systems capable of running arbitrary rules and policies. In essence, smart contracts are nothing more than user-defined code that can be triggered by peers to allow the distributed execution of arbitrary state transition functions between network nodes, enabling mutually distrustful peers to interact with each other in a transparent and deterministic way, since everyone has access to the same code [16].

For example, one can implement a smart contract that is able to support the lifecycle of a passenger's flight insurance. The passenger can register a transaction in the blockchain representing the payment of the insurance and the smart contract code can process the business logic that handles gathering information about that flight from external services. Depending on that same information, if the flight is on time or delayed, the contract can deterministically decide if the passenger will pay or get paid the insurance.

The first successful implementation of smart contracts was made by Ethereum [12], with other systems following soon after, as we will see on the next section (Section 2.4). In Ethereum, smart contracts execute in a sandboxed runtime environment called the Ethereum Virtual Machine (EVM), which is replicated across every network node. The contract code is persisted on the blockchain and deployed within a block, similarly to a set of transactions, and can be triggered by sending transactions towards its address [16], updating each node's state through a State Machine Replication (SMR) process. Contracts are regulated by special fees (*gas* units) that ensure code termination and play a big part in transaction verification, since every node locally verifies contract code conformance with gas limits.

2.3.4 Blockchain Design Issues

Blockchains are certainly fascinating in their own design and properties, but not without its caveats, namely, low scalability and lack of privacy.

Low scalability. The blockchain's low scalability manifests itself in limited throughput and high confirmation latency of transactions, high bootstrap time for new nodes, and significantly elevated costs regarding storage requirements, network bandwidth and CPU usage [20]. This severely impacts the adoption rate of blockchain technology and its application to other use cases beyond cryptocurrency. For a better understanding of the problem, we will follow the vision of Croman et al. [20] and unfold the blockchain into

different planes, each with notable inefficiencies: Network, Consensus, Storage and View. Given the scope of this thesis, we won't consider the Side plane defined by the authors.

The *Network plane* refers to the message exchange between peers during the several phases of the protocol. The inefficiency related to this plane lies with the local validation of transactions by each node, which delays transaction propagation, and with duplicate transmission, since each transaction is propagated twice: first when a node proposes a transaction for the first time and once again when a block is mined and relayed.

The *Consensus plane* concerns itself with improving the speed of the consensus algorithm of the blockchain and reducing block latency. Network-wide proof-of-work consensus, while completely decentralized, is a computationally costly and time-consuming algorithm, standing as the major bottleneck to throughput [74]. There are several possible approaches for this plane, from sharding peers into groups and reconciling consensus by means of some Byzantine Fault-Tolerant (BFT) SMR protocol, such as PBFT [15], to semi-decentralizing the blockchain in a hierarchical fashion with a top-level blockchain coordinating several smaller instance blockchains (sidechains), resorting to consortium consensus protocols, thereby delegating some level of trust in small sets of trustable entities running a PBFT-based consensus, or even proof-of-stake, a protocol in which nodes vote with whatever digital tokens they own on the system (a stake), instead of CPU power.

The *Storage plane* describes on how the ledger and state data are persisted across nodes. In its original specification, every node of the blockchain holds a full copy of the ledger. While this allows for a truly decentralized way of verifying transactions and blocks, it brings a heavy storage cost given the growth potential of the blockchain[2]. A possible approach is to shard the ledger across nodes and use a distributed hash table, or a similar structure, to identify which nodes hold which part of the ledger.

Lastly, the *View plane* addresses how peers view the current state of the system. Consequently, this plane is also related with bootstrapping, the process by which newly joining nodes get up-to-date on the system's state. The bootstrapping protocol is a cumbersome SMR process that requires new nodes to download all of the blockchain down to the genesis block and process the entire transaction history before being able participate in the system. Most systems approach this problem with the introduction of partial nodes – nodes that do not possess the ledger and are able to broadcast and receive transactions but cannot verify them for themselves, relying on other nodes they trust for that matter [33, 54]. More efficient approaches can be applied by relaxing the trust model restrictions of the blockchain and outsourcing views in a cryptographically-secure and authenticated manner from a subset of nodes [20, 69].

Lack of privacy. The privacy problem surfaces as the trade-off with blockchain's transparency and auditability. At the price of permanently recording every transaction publicly, sensitive information can be disclosed to unwanted parties, breaking confidentiality, and public-keys can be traced to identify traffic patterns, breaking anonymity.

[2]Bitcoin's blockchain size over time: https://blockchain.info/charts/blocks-size

Addressing *transaction confidentiality* is usually done by encrypting information, isolating it in private channels between authorized parties or a mix of both. The Hawk framework [49], for example, allows for the development of private smart contracts through the use of public-key cryptography and non-interactive zero-knowledge proofs (zk-SNARKs) – cryptographic schemes in which a message can be proven valid without any further information other than the fact that it is valid being conveyed and without peer interaction. Zerocash [5] does a similar approach, obfuscating transaction data that can only be accessed by the parties involved in a transaction by harnessing zk-SNARKs as well. Another solution is to split blockchain traffic into different isolated chains so that information is revealed only to authorized nodes within the context of a single chain [28].

Regarding *peer anonymity*, plausible approaches such as allowing nodes to transact under different public-keys [33] or splitting blockchain traffic into different subnets can frustrate attempts to associate a node with a single public-key. More elaborate proposals, such as the ones by Xu et al. [78] and Heilman et al. [35], consist of systems in which virtual intermediaries, available to participating nodes in a transaction, execute smart contracts between them, concealing the true identity of the participants. Other noteworthy advances, such as Zerocash [5], resort again to zk-SNARKs to prove the validity of transactions without the need of peers knowing the identity the node that proposed them.

2.4 Blockchain Platforms

In the previous chapter, we have mentioned a concept called a *blockchain platform*. A blockchain platform is a system fundamentally based on blockchain technology that eases the process of building decentralized applications by providing a foundational layer to build upon. Despite the infancy of blockchain technology, an increasing number of platforms is emerging quickly, with several examples of decentralized applications built upon these platforms in healthcare, supply chains, uncensorable social media, automated locks, and many others use-cases [1, 25, 46, 77].

In this section, we will iterate over a representative set of platforms that can be leveraged for our objective. Unintentionally, this set is composed only by open-source platforms. In the end, we will be able to discuss on the benefits and drawbacks of each system by pondering on the following sets of characteristics:

- *Software Engineering*: Relates to the software development and construction characteristics of the system:
  - Quality of the resources documenting each system component;
  - Size and activeness of the community behind the system. For a consistent evaluation of each community, we will resort to Github metrics: *stars* – which we interpret as a form of user appreciation – and *forks* – which are the result of users reusing the platform's source-code to build new systems.

- *System Architecture*: Describes the architecture of the system having into account the scalability and privacy design issues of the blockchain described in Subsection 2.3.4:
  - Whether the system is permissioned or permissionless. Permissioned blockchains impose restrictions on the actions peers are allowed to execute (participating in consensus, validating transactions and blocks, executing smart contracts, etc.), while permissionless blockchains do not;
  - Degree of decentralization of the system taking into account dependencies on third-parties, external services or federated groups of nodes. Decentralization defines to what extent may a dependency influence the system into behaving erratically and present a single point of failure for the entire system;
  - Support for orchestrating multiple blockchains under a single platform;
  - Transaction privacy and peer anonymity guarantees, in order to avoid tracing and disclosure of sensitive information and traffic patterns;
  - Transaction flow, in comparison with the original blockchain specification;
  - Support for Byzantine fault-tolerance (BFT) and how many nodes are needed to tolerate f faulty replicas;
  - Consensus mechanism being employed by the system. Whether it is pluggable (i.e. changeable between BFT/crash-only) and what's the system's throughput scalability with respect to transactions per second (tps);
  - How is state and ledger data replicated across network nodes: globally across the network or partially across subsets of nodes;
  - How do nodes update their system's state view, with regards to the consistency model being used as well as on the ordering of messages;
- *Expressiveness and Programming Support*: Relates to the programming model that the system provides for the implementation of custom logic. We will also address smart contract extensibility, since some platforms implement only a partial notion of smart contracts and do not support extensible content-agnostic code to be executed in the blockchain.

2.4.1 Ethereum

Ethereum [12] is the best known permissionless blockchain system after Bitcoin, with a rather active and large community (around 10000 stars and 3000 forks on Github). It is also an open-source platform with a very complete and descriptive documentation. Its creation was largely motivated by the idea of improving upon Bitcoin's original architecture by retrofitting the blockchain with a Turing-complete scripting mechanism that would allow custom code execution, i.e. extensible smart contracts.

At the time of writing, Ethereum implements proof-of-work consensus mechanism, ensuring a BFT decentralized agreement between all network nodes assuming 51% of them are honest. It is planned to transition to a hybrid proof-of-work/proof-of-stake model in the near future. Albeit consensus is optimized to be memory-hard instead of CPU-hard, Ethereum suffers from the same low throughput capacity as Bitcoin does, reaching an average maximum of 15 tps. Likewise, as in Bitcoin, every node keeps an entire copy of the ledger and updates its system state view by executing a causally ordered and eventually consistent SMR process. The transaction flow of Ethereum is roughly the same as Bitcoin. Additionally, since the platform is intended at a public setting, Ethereum does not provide any transaction privacy or peer anonymity guarantees.

Being an open-source platform, Ethereum gave birth to multiple other platforms. We will address two well-known examples in the blockchain community, both of them open-source, that were forked from the Ethereum codebase: Quorum and Hydrachain.

2.4.2 Quorum

JPMorgan's **Quorum** [56] was built on the premise of creating a enterprise-oriented blockchain, with Ethereum chosen as the groundwork for such platform to grow upon. Accordingly, Quorum inherits the EVM and smart contract extensibility from its precursor.

Essentially, Quorum is a permissioned blockchain divided into a public and a private network. As with regular blockchain systems, participants process all public transactions received from the public network. However, the transactions in the private network rely on proxy agents similar to [78], and asymmetrical encryption of data to ensure that only the parties involved in a private transaction can process it. Private smart contracts are also made possible by segmenting contract storage across nodes. Quorum further introduces a degree of anonymity into the system by leveraging zk-SNARKs [5] to shield private smart contract information between peers.

Quorum's consensus mechanism is consortium-based [20] and has two distinct implementations: Istanbul BFT (able to tolerate f faults with a population of $3f + 1$ nodes) and Raft (does not support BFT in exchange for higher throughput). There are no official performance metrics available, but throughput is presumably highly scalable in any of the implementations (with an average maximum throughput of over 1000 tps on standard conditions) if we extrapolate the results of other platforms that also execute consensus in a consortium PBFT-like manner. Regarding view computation, this consensus approach ensures a strongly consistent, totally ordered SMR across the network.

Overall, Quorum is a very privacy-focused platform. JPMorgan itself and contributors from the open-source community seem to be actively supporting the platform (which has around 1500 stars and 300 forks on Github). The documentation provided is very rich and thorough on most aspects of its architecture. However, Quorum is heavily dependent on JPMorgan's Constellation network and on a federated group of nodes for consensus, making it only partially decentralized.

2.4.3 Hydrachain

Hydrachain [40], an open-source community-driven fork, introduces permissioned ledgering into Ethereum, similarly to JPMorgan's Quorum. Hydrachain's consensus mechanism is inspired by a consensus engine known as Tendermint [71] and is a BFT SMR algorithm, executed in a consortium-like fashion, tolerant up to the order of $3f + 1$ with f faulty processes. As with Quorum, there are no proper performance metrics available, but since consensus is executed in a small subset of nodes and presumably approximates Tendermint's algorithm, its throughput scalability is presumably high as well, with over 1000 tps. View computation is a strongly consistent, totally ordered SMR process.

Perhaps one important feature of Hydrachain is its ability to bypass the EVM, allowing the development of native smart contracts (written in Python) with faster execution times. However, this allows for undeterministic code to be written by naive developers, deadlocking the entire network. The platform was also designed with the architectural idea of supporting multiple chains running in parallel, in which each node could participate concurrently in two or more chains at once, allowing both context and privacy concerns to be isolated. However, this feature never got past being future work and Hydrachain's small community (with around 250 stars and 100 forks on Github) seems to have gone cold, with no new features added into the platform's source-code over a year and with a very poor documentation which provides little insight into its architecture.

Other than the described features, Hydrachain is currently very similar to Ethereum and even maintains an EVM for compatibility purposes with its predecessor.

2.4.4 The Hyperledger Project

The Hyperledger project consists of a plethora of different open-source blockchain platforms and tools created by several contributors, aiming to provide standard cross-industry collaborative tools for building rich decentralized applications [28]. For the purpose of this thesis, we will be focusing on its main platforms: Fabric, Sawtooth, Burrow and Iroha.

Fabric (also known as HLF) [42] was the kickstarter for Hyperledger, being developed by IBM. We will only address the V1 version of the platform, since it is the most recent one. Its community is rather large, with around 2500 stars and 2000 forks on Github, and contributors seem to be actively engaged into improving the platform. Its documentation is also very detailed and goes into depth of almost every aspect of its architecture.

Fabric supports the notion of multiple permissioned and interoperable chains, splitting sets of nodes across *channels*, in which nodes can participate concurrently. This allows partitioning smart contracts and transactions accordingly, isolating data confidentiality concerns. Ledger replication is partial, since each ledger is maintained and shared only between authorized nodes on a per-channel basis. At the present moment, Fabric does not leverage any anonymity mechanisms. Regarding consensus, Fabric resorts to an external off-chain ordering service that establishes total order on the transactions of all

its chains. This service is pluggable, but its official implementation is crash fault-tolerant only. While this approach allows a unified consensus between chains and effectively enhances throughput, yielding over 1000 tps, it centralizes consensus by delegating trust to the ordering service. It can be made BFT using the work by Sousa et al. [6].

The transaction flow within Fabric differs radically from the original blockchain specification, as illustrated in Figure 2.3. In Fabric, within the context of a channel, clients propose transactions to a subset of nodes called *endorsers*. In turn, these nodes are responsible for verifying the validity of transactions and the sender's signature, signing (endorsing) the transaction and returning it to the client, which is responsible for sending the now endorsed transaction to the consensus service. After consensus is reached, the transaction is then propagated within a block to the *channel* where the nodes are participating, and their ledgers and view of system's state updated, in a strongly consistent, totally ordered SMR process.

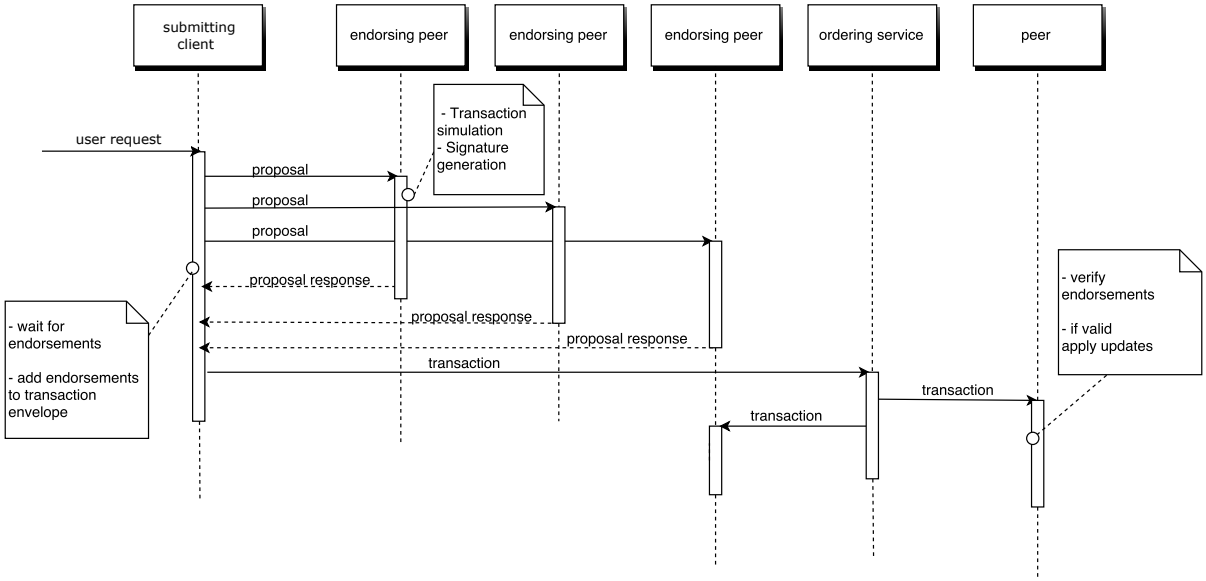

Figure 2.3: Hyperledger Fabric transaction flow (adapted from [69])

One last important thing to note about Fabric is Chaincode, its implementation of smart contracts. Chaincode follows closely the architecture of the EVM, providing a sandboxed runtime environment for each node which interprets contracts written in Go, but does not have a built-in *gas limit* that limits execution of contract code. Despite the implementation being different, Chaincode allows for generic and extensible smart contracts, identically to the EVM.

Sawtooth [44] was developed by Intel, with a strong focus on increasing the blockchain's consensus efficiency and minimizing resource consumption. Its community is small yet active (around 500 stars and 250 forks on Github), and it has a rich documentation describing its architecture. To implement a permissioned system, Sawtooth borrows a modified EVM from Burrow, which we will describe ahead. Its consensus mechanism is

pluggable, allowing for a crash fault-tolerant random leader election protocol, but it defaults to a novel approach dubbed *proof-of-elapsed-time*, which relies on nodes requesting wait times from Intel's SGX CPU enclaves – trusted execution environments that allow code to be run in tamper-resistant private areas of the CPU – to elect a leader. The node with the shortest wait time gets to add its block to the blockchain without expending CPU cycles mining. Its support for BFT is the same as proof-of-work, requiring a majority of honest nodes (51%) in order to tolerate faulty nodes, and its throughput scalability is presumably high (over 1000 tps).

Regarding transaction flow, ledger replication and view computation, Sawtooth behaves identically to Ethereum or Bitcoin. Similarly, it does not address privacy and anonymity as of yet. The degree of decentralization of the system merely depends on its nodes possessing Intel SGX technology, since every other protocol action is decentralized.

Burrow [41] was previously known in the community as Eris:db, a blockchain platform forked from Ethereum. As with Sawtooth, its community is small but growing (currently around 200 stars and 100 forks on Github) and actively contributing to the platform's development. Its documentation is mediocre: it gives an overall understanding of the features of the system but lacks in detailing them. Being an Ethereum fork, Burrow makes use of a modified EVM featuring an access control layer implemented directly on top of it as to provide a permissioned runtime environment for extensible smart contracts.

Burrow's consensus mechanism is consortium-based and implemented by the Tendermint engine [71], allowing for a highly scalable throughput (reaching over of 1000 tps) and BFT with $3f + 1$ nodes in the presence of f faults. Nodes update their view of the system through SMR, ensuring strong consistency and total order. Like Fabric, Burrow's architecture also allows for multiple chains, but at the moment consensus is established at chain level and interoperability is limited, with plans of supporting this functionality in the near future. This architecture somewhat allows for a weak transaction privacy guarantee in the sense that data is confined within each chain. However, unlike Fabric, where nodes can participate in multiple *channels* and establish private ledgers with other peers, here nodes cannot participate concurrently in several chains. Anonymity is also not addressed by this platform.

Iroha [43] is a permissioned blockchain that emphasizes on mobile device support with simplicity in mind, providing a bare-bones platform to build upon with mobile development libraries included. At the heart of its consensus mechanism lies a BFT consortium-based consensus algorithm, dubbed Sumeragi, that establishes consensus at transaction level and takes into account a peer reputation protocol to choose the order of nodes that get to process transactions, able to tolerate f faults with over $3f + 1$ nodes. Its throughput is presumably high (estimated over 1000 tps), ledger replication is done globally (network-wide) and it ensures a strongly consistent, totally ordered SMR view computation. It does not provide privacy or anonymity guarantees in its current version.

As with other Hyperledger platforms, with the exception of Fabric, Iroha has a small yet active community (around 500 stars and 150 forks on Github). Its documentation is,

however, rather poor. Iroha also implements its own version of smart contracts – *chaincode* – written in Java and executed against the Java Virtual Machine (JVM) on sandboxed environments present in each node. As with EVM contracts, these are also extensible.

2.4.5 Counterparty and Bitcoin

Bitcoin, despite not being intended as a platform, has long gained community interest for being the original proponent of blockchain technology and its most successful use-case. Purposefully implemented limitations of Bitcoin due to security implications made it a poor candidate for anything other than cryptocurrency, but interest to extend it prevailed.

Enter **Counterparty** [19], an open-source platform built upon Bitcoin, enabling smart contracts to be run on Bitcoin's blockchain by borrowing the EVM. In broad terms, Counterparty shares exactly the same blockchain as Bitcoin, even piggy-backing on its proof-of-work consensus, but smart contracts can only be interpreted by Counterparty.

Despite this added functionality, this platform's architecture is the original architecture of Bitcoin, and it can almost be seen as a smart contract layer. Therefore it is a permissionless and completely decentralized blockchain platform that suffers from the same scalability and privacy issues as its predecessor. Its documentation is actually very complete and it has a small but active community working on the project, with around 200 stars and 150 forks on Github.

2.4.6 Multichain

The **Multichain** platform [33], as the name suggests, was built to support multiple permissioned chains running in parallel. Like Burrow, consensus is run at blockchain level and not in a unified service for all chains. However, Multichain supports chain interoperability, and transactions in one chain can be used to trigger transactions in the other.

Multichain's consensus is consortium-based, restricted to a subset of nodes that execute a custom crash fault-tolerant algorithm in which a single leader is selected on a round-robin policy for each block being added to a chain. This ensures fairness and a high throughput scalability, achieving over 1000 tps on standard conditions, but does not provide BFT and partially centralizes consensus. Nodes update their system view in an eventually consistent, totally ordered SMR process. Transaction confidentiality is retained by encrypting data exchanged between peers through public-key cryptography and by isolating different private contexts into different chains. Regarding anonymity, Multichain employs a mechanism that allows each peer to transact using different public-keys in an attempt to frustrate an attacker into tracing a known key.

The community behind Multichain is relatively small (around 300 stars and 100 forks on Hithub) but active. The project is open-source and its documentation is of mediocre quality, providing high level insight of the system. The downside of this platform is the inability of providing the developer a programming model to develop custom policies and rules. Multichain does not support smart contracts, making its native code inextensible.

2.4.7 Summary

Tables 2.1 and 2.2 summarize the properties of each platform according to the characteristics highlighted in the beginning of this section.

	Software Engineering				Expressiveness and Programming Support	
	Open source	**Documentation quality**	**Impact/Community (appr. Github metrics)**	**Active**	**Programming model**	**Extensible smart contracts**
Ethereum	✓	Rich	Large (10000 stars, 3000 forks)	✓	EVM smart contracts	✓
Quorum	✓	Rich	Medium (1500 stars, 300 forks)	✓	EVM smart contracts	✓
Hydrachain	✓	Poor	Small (250 stars, 100 forks)	✗	EVM smart contracts or native Python	✓
Hyperledger Fabric V1	✓	Rich	Large (2500 stars, 2000 forks)	✓	Go chaincode over Docker	✓
Hyperledger Sawtooth	✓	Rich	Small (500 stars, 250 forks)	✓	EVM smart contracts	✓
Hyperledger Burrow	✓	Mediocre	Small (200 stars, 100 forks)	✓	EVM smart contracts	✓
Hyperledger Iroha	✓	Poor	Small (500 stars, 150 forks)	✓	JVM chaincode	✓
Counterparty	✓	Rich	Small (200 stars, 150 forks)	✓	EVM smart contracts	✓
Multichain	✓	Mediocre	Small (300 stars, 100 forks)	✓	Not supported	✗

Table 2.1: Comparison of blockchain platforms: *Software Engineering* and *Expressiveness and Programming Support*

From Table 2.1, we can infer that most of these platforms, despite documentation issues, can be successfully used to build decentralized applications. Two platforms stand out for their flaws, namely, Hydrachain and Multichain. Hydrachain seems to have lost momentum from its contributors and community, and Multichain does not accept a generic programming model useful enough for implementing custom business logic, losing interest for anything other than financial use-cases.

From Table 2.2, we can clearly see a correlation between consensus mechanisms, decentralization and throughput scalability: platforms using proof-of-work, while benefiting from a fully decentralized consensus, suffer from low throughput (Ethereum and Counterparty). Other platforms override the original proof-of-work mechanism with a federated group of nodes or an off-chain service, resulting in partial decentralization but better throughput. In terms of privacy and anonymity, most platforms do not provide any guarantees, while HLF and Burrow provide transaction privacy (although Burrow's approach is simply separating data in different non-interoperable chains and therefore inferior to Fabric's), and Quorum and Multichain provide both transaction privacy and anonymity. Pluggable consensus is supported only by HLF, Quorum and Sawtooth. Overall, the most complete platforms that can be used to build dependable privacy-concerned decentralized applications able to yield a high throughput are Quorum and Fabric.

	Type	Decentr.	Mult. chains	Network plane: Transaction flow	Consensus plane: Mechanism	Plug.	BFT	Throughput scalability	Storage plane: Ledger replication	View plane: View computation	Tx. privacy	Peer anonym.
Ethereum	Permissionless	Full	✗	Bitcoin-like protocol	PoW (migrating to hybrid PoW/PoS)	✗	51 %	Low (< 100 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗
Quorum	Permissioned	Partial	✗	Private transactions propagated in a subset of nodes, while public transactions are network-wide	Consortium	✓	3f+1[1]	Presumably high (> 1000 tps)	Partial (private smart contracts are only replicated between authorized peers)	Strongly consistent, totally ordered SMR	✓	✓
Hydrachain	Permissioned	Partial	✗	Transactions are submitted to consortium consensus, blocks are then propagated network-wide.	Consortium	✗	3f+1	Presumably high (> 1000 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗
Hyperledger Fabric V1	Permissioned	Partial	✓	Transactions are proposed to a subset of nodes (endorsers), then to the external consensus service, and finally to corresponding *channels*.	Off-chain service	✓	3f+1[2]	High (> 1000 tps)	Partial (each *channel* holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✓	✗
Hyperledger Sawtooth	Permissioned	Full[3]	✗	Bitcoin-like protocol	PoET / random leader election	✓	51%[4]	Presumably high (> 1000 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗
Hyperledger Burrow	Permissioned	Partial	✓-	Transactions are submitted to consortium consensus, blocks are then propagated network-wide.	Consortium	✗	3f+1	High (> 1000 tps)	Partial (each chain holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✓-	✗
Hyperledger Iroha	Permissioned	Partial	✗	Transactions are submitted to consortium consensus, blocks are then propagated network-wide.	Consortium	✗	3f+1	Presumably high (> 1000 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗
Counterparty	Permissionless	Full	✗	Bitcoin protocol	PoW	✗	51 %	Low (< 100 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗
Multichain	Permissioned	Partial	✓	Transactions are submitted to consortium consensus, blocks are then propagated network-wide.	Consortium	✗	Not BFT	High (> 1000 tps)	Partial (each chain holds a ledger between a subset of nodes)	Eventually consistent, totally ordered SMR	✓	✓

[1] With Istanbul BFT; [2] With the unofficial BFT-SMaRt ordering service presented in [6]; [3] Assuming every node is equipped with Intel's SGX technology. Otherwise decentralization is partial;
[4] If using PoET.

PoW = Proof-of-work, a protocol where peer nodes *mine*, i.e. expend CPU power, to solve a cryptographic problem in order to achieve consensus
PoS = Proof-of-stake, an alternative protocol to PoW in which nodes vote with a stake of digital tokens they own on the system, rather than investing CPU power
PoET = Proof-of-elapsed-time, a lottery-like protocol that relies on peer nodes requesting wait times from Intel SGX CPU enclaves to elect a consensus leader based on the shortest wait time

Table 2.2: Comparison of blockchain platforms: *System Architecture*

2.5 Threshold Signatures for Blockchain Transactions

In this section, we enter into detail on a group-oriented cryptographic construction that can be leveraged for our objective – threshold signatures – with the intent of providing cooperative and decentralized transaction signing and verification processes into what we understand to be the traditional blockchain architecture. First, we discuss threshold signatures definitions and current applications. Then, we discuss on the possible use, advantages and eventual obstacles of their use in a blockchain.

2.5.1 Threshold Signatures

Threshold signatures (or (t,n)-Threshold Cryptographic Digital Signatures) [11, 21, 22] are cryptographic constructions that enable the distribution of secret information (i.e. secret keys or secret key shares) and digital signature computation (generation and decryption for verification) between a number of parties n, in order to remove a single point of failure or the dependency of a single point of trust within a system. The goal is to allow any subset of more than t parties to jointly reconstruct a secret or perform the required computation while preserving security, even in the presence of an active adversary, which can corrupt up to t (threshold) parties. The security notion for threshold signatures requires that no polynomial-time adversary that corrupts any t parties can learn any information about the secret key or forge a valid signature on a new message of its choice.

In order to be able to build a threshold signature, a secret key is broken apart into secret shares and distributed among n parties, with the help of a trusted dealer or by running an interactive protocol among all involved parties. To sign a message M, any subset of more than t parties can use their shares of the secret and execute an interactive signature generation protocol. The output of such protocol is a signature of M that can be verified by any peer, using an unique fixed public-key corresponding to the above shares.

An important property of any threshold signature scheme is robustness. Robustness requires that even t malicious parties that deviate from the protocol cannot prevent it from generating a recognizable valid signature, as shown in Figure 2.4. Another useful property of threshold signature schemes is proactiveness. Proactiveness relates to the possibility of periodic refreshment of secret shares, to protect a system from an adversary that builds-up knowledge of a secret by several attempted progressive break-ins to several locations. In general, the main goals of threshold signature constructions are to provably achieve the following properties: i) to support as high a threshold t as possible; ii) to decentralize digital signature generation and verification; iii) to be robust and useful; iv) to allow the use as a proactive scheme; and v) to be efficient enough in terms of computation, interaction and length of the shares.

In the related work of proposals for threshold signature schemes, we can find several

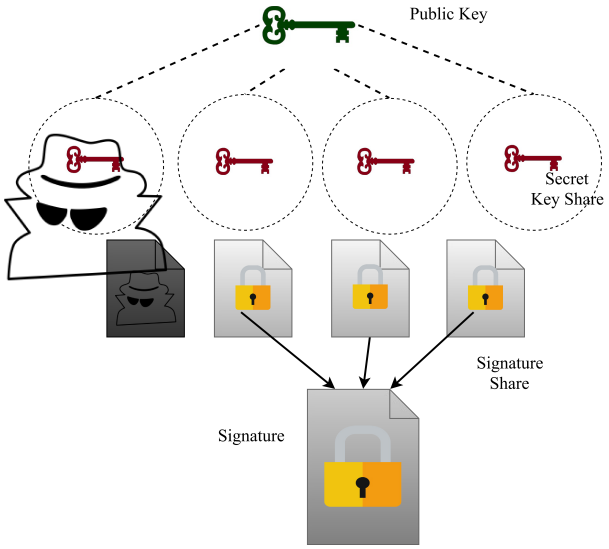

Figure 2.4: Message signing via a threshold signature sheme (from [69])

relevant contributions [22, 23, 29, 30, 34, 57, 65]. Despite the fact that some of these contributions are more particularly concerned with theoretical foundations, the proposals of [21, 34] lack security proofs, and the schemes of [22, 23] are considered as non-robust, while those of [29, 57] are considered until today as robust and proactive schemes. However, it is well known that such schemes are always sources of a high amount of interaction, and must be addressed carefully for scalability purposes. For a more practical approach in the domain of our thesis, we focus on threshold Digital Signature Standard (DSS) signatures, proposed in [30], and on robust threshold RSA signatures, proposed in [65], since both are recognizably robust and have well-known practical implementations.

The threshold DSS signature in [30] is theoretically considered robust and does not require a trusted dealer. It consists of a proof of security without the random oracle assumption. The proposal deals with technical difficulties such as combining shares of two secrets into shares of the product of these secrets and producing shares of a reciprocal of a secret given shares of this secret. To achieve robustness, this type of threshold signature uses the error-correction techniques of Berlekamp and Welch [75]. As a result, the threshold DSS can only tolerate $t < n/4$ malicious parties, requires a lot of interaction and the complexity increases considerably related to the base signature scheme. The scheme can be made proactive following the methods of [36, 37].

The robust threshold RSA signature scheme proposal [65] is proven secure in the random oracle model. It can tolerate $t < n/2$ malicious parties and the signature generation algorithm is non-interactive. This scheme requires a trusted dealer to run the key generation protocol. The public-key uses an RSA modulus that is a product of two safe primes and the protocol utilizes zero-knowledge proofs in the random oracle model in order to achieve robustness. Proactiveness is not considered in [65], even that it can be addressed, orthogonally.

2.5.2 Applications of Threshold Signatures

Distributed trust has been a key motivation for threshold signatures. Shamir [65] argued early in 1979 about the relevance of threshold cryptography for key management and distribution protocols. Storing a key in a single location is not robust, but keeping multiple copies of the same key opens the possibility for security breaches. Overall, no single entity should be trusted to keep the secret key for a particular signature.

For instance, in [58, 80], the introduction of threshold cryptography is motivated by the replication of distributed services in a way that preserves the availability and correctness properties of the system and maintains causality of requests, even if several replicas are corrupted. In the specific context of [58], threshold cryptography allows a client to maintain one public-key for the replicated service, instead of one public-key for each replica. The client does not need more storage and does not undergo a higher computational cost than in the case of a non-replicated service, regarding authenticated messages from a transparent trustable peer-group membership of distributed processes.

Following the same idea of [58], the motivation in [13] is to define an architecture for distributing trusted services in a fully asynchronous environment supported over the Internet. This architecture includes fully decentralized certification authorities (CA), distributed secure directories, or, in more concrete examples, fully decentralized and trustable DNS services. The advantages of threshold signatures for DNS are also further addressed in [14]. DNS Security Extensions (DNSSEC) use a technique called zone signing to provide authentication for replies from the DNS name resolution service, with the private key for signing the zone stored locally in some entity in the network. This single entity introduces reliability and security issues as it is a single point of failure. To tackle this issue, the proposal of [14] is to use a threshold RSA signature scheme to securely replicate the authoritative servers in the context of zone signing replies.

2.5.3 Decentralized Blockchain Transactions and Verification

In summary, threshold cryptography is a powerful tool that has been widely explored as a more secure approach for service replication. We believe that the use of this tool in blockchain platforms will be a relevant step in achieving fully decentralized ledgering and reducing trustability assumptions. In fact, Goldfeder et al. [32] already discussed the possible advantages of leveraging threshold signatures for Bitcoin wallets.

The design and implementation of threshold signatures in a blockchain ecosystem must be carefully addressed to provide the required balance for scalability, performance and improved decentralization guarantees. However, this must be designed and implemented without sacrificing throughput or latency conditions. To do so, on one hand, we can restrict membership control conditions in order that only a subgroup of nodes will be involved in signing and validation processes, also considering that we can have different subgroups involved. On the other hand, we can use implementation schemes for threshold signatures using lightweight cryptographic primitives, such as ECDSA-based

threshold signatures or possible hardware implementations at least in certain nodes of the blockchain.

Another option is to follow the ongoing research on the optimization and efficiency of threshold schemes for multiparty computations [8]. Finally, another relevant design option is to explore synchronized forms of aggregated signatures (or aggregated multi-signatures – ASM), minimizing the number of rounds for verification processes, as proposed for the optimization of BLS multi-signatures [9]. These constructions support both signature compression and public-key aggregation, allowing to verify that a number of parties signed a common message m. The verifier only needs a short multi-signature, a short aggregation of their public keys, and the message m. In [9], the authors present constructions that are derived from Schnorr signatures and from BLS signatures, and the possible adoption in a blockchain. With such ASM schemes any subset S of a set of n parties can sign a message m so that a valid signature discloses which subset generated the signature (hence the subset S is accountable for signing m). ASM schemes can achieve signature sizes of only $O(k)$ bits over the description of S, where k is the security parameter. Similarly, the aggregated public-key is only $O(k)$ bits, and is completely independent of n.

For the context of our thesis, the lack of availability of those recent schemes in tangible implementations is an issue. Thus, an interesting practical would be to use generic models for threshold signatures schemes while bearing modularity and extensibility design assumptions in mind. This would allow the possible use of cryptographic providers implementing such schemes (e.g. through libraries libraries) in a pluggable and possible object-factory implementation in the future, without compromising other system model assumptions. Such approach must tackle the way how group signatures can be mapped together with Byzantine fault-tolerance properties and the used mechanisms at the consensus plane level of the blockchain platform at use. This is one relevant direction in our proposed solution and implementation.

In [69], the authors present a study of the practical use of threshold signatures, namely threshold RSA [65] and threshold BLS, for the HLF [42]. Both are non-interactive and deterministic schemes. The authors concluded that leveraging one of such schemes for Fabric does not cause any relevant negative impact on the system, therefore being equivalent to the non-threshold version of the scheme and allowing for a seamless integration. The authors also expressed some concerns, performance-wise, in anticipating the use of RSA threshold signatures, which are more efficient than BLS according to the authors, alternatively to ECDSA multi-signatures, since the use of ECDSA multi-signatures can scale better for an increasing number of nodes. A final relevant point is that in threshold signature schemes a validator needs to validate only one signature, whereas for multi-signatures the number of the signatures that must be validated increases linearly with the number of participants. Identically, threshold signatures always have a fixed size, while multi-signatures vary with the number of signers.

2.6 Critical Analysis

We believe that current IoT architectures need to be revisited and that blockchains can provide truly resilient systems with no single point of failure. However, we find it unlikely that blockchain technology will integrally replace the role of clouds as part of IoT architectures but will rather work as an add-in for decentralized ledgering that can be used to ensure privacy, robustness and security for every endpoint. The architecture methodologies presented by [24, 64] are a relevant point for our goals, since they present blockchains as an intermediary layer between IoT devices and the cloud and remove the weight of the blockchain protocol away from the IoT, as well as the edge computing models proposed in [10] to provide better scalability and privacy for IoT systems and give back some control to the users while reducing stress on cloud services.

We intend to stand our research upon an existing platform that emphasizes on high throughput scalability, possibility for BFT, confidentiality and decentralization; yet, we wish to retain the ability of our system model to be agnostic to a permissioned blockchain platform implementation. Our justification for these characteristics is that an ideal decentralized IoT system is supposed to behave in near real-time, while still being robust against potential adversaries and preserving privacy. We require consensus pluggability for benchmarking performance with BFT and non-BFT consensus in combination with threshold signature schemes for transaction verification and accept on compromising with a partially decentralized platform by prioritizing throughput scalability in detriment of full decentralization. Moreover, the use of open-source blockchains, addressing a modular permissioned model designed to support pluggable implementations of different components, such as the consensus and membership services as two separated concerns, is an important consideration for building an extensible decentralized ledgering platform. These requirements are what guides our system model as to allow any generic base blockchain platform that assures them to be used for a blockchain-enabled IoT architecture.

However, for our implementation we need a concrete blockchain platform that implements such requirements. From our summary in Section 2.4, our best candidates would be Quorum or HLF. Both improve over the original transaction flow and ledger replication of the blockchain, whilst providing strong consistency with a totally ordered message delivery. However, our decision was HLF for a few reasons: *i*) it provides the notion of channels and multiple chains, which can be used to isolate transactions between several peers in a straightforward way. Quorum is only able to split traffic between a private and a public network; *ii*) it has a larger community than Quorum and finds more academic and scientific usage than other platforms [69]; and *iii*) despite not providing anonymity as Quorum does, we find that this property is more relevant in scenarios where public transactions exist. Since Fabric is permissioned and allows for private channels to be set up, this property becomes irrelevant for our case. We also find that it is possible to

execute the HLF's off-chain consensus mechanism without depending on a single third-party and risk centralizing the network, as a consortium of entities responsible for the decentralized management of the ordering service can be established.

Our idea is to follow on the discussion of [69] and add a fully decentralized group-oriented signature verification process of transactions and smart contracts, integrating threshold signatures as a base component of a given blockchain platform and its supported services, providing participant nodes with the functionality to generate and verify, in a distributed manner, a single fault-tolerant signature. Nevertheless, we believe that the definition of which signature scheme to use for a given set of transactions should be dynamic and decided by the participating entities interacting between themselves and that multiple schemes can be harnessed by a single platform as to adapt to multiple use-cases and scenarios of the IoT paradigm. The ability of specifying and allowing participants: users, service providers, IoT devices and even edge-based smart hubs [10], to scrutinize such system properties – which signature schemes will be used and which nodes will endorse a given transaction – is a powerful and interesting concept, and one that can be leveraged through the use of a smart contract specification. There is also the need to to assess the throughput of a blockchain platform assuring both decentralized trustability assumptions and Byzantine fault-tolerant consensus guarantees. Further, it is vital to compare the performance of threshold signature schemes with existing multi-signature ones, taking into account scalability concerns related to peer interaction and signature size, which can be done through the use of a smart contract specification for signature verification that implements both schemes.

Chapter 3

System Model and Architecture

In this chapter we present our system model proposal for a decentralized and scalable ledgering system that can be built upon an existing blockchain platform with extensible smart contracts support. The idea is to address a model and foundations for a blockchain-enabled IoT platform. We start by providing an application scenario for an easier comprehension on the applications of our system model, then we will go into its details, architecture and corresponding software components, and finally present some considerations regarding the threat model of the system and expected runtime behaviour.

3.1 Application Scenario

We can imagine a scenario to better clarify the addressed system model presented next. In our scenario, a user requests a vacation rental through a provider that offers such service, like Airbnb. Likewise, imagine that every apartment in this fictitious rental service is equipped with smart things – smart door locks, smart lighting, smart appliances – connected to a smart hub that requires a confirmation from the service stating that the user which intends to access the apartment has payed the price of the rental (and in the correct amount) in order to comply with his commands.

To rent an apartment, a user uses a smartphone linked to his credit card or his bank account and executes a transaction between him and the service. The service verifies the transaction and returns an identifier for the user to present to the apartment's smart hub, which can communicate directly with the user's smartphone and with the rental service. The user then presents its identifier to the hub, which in turn contacts the service to confirm the identifier's validity. If it checks out, the user's identity is proven and he's given permission to use the apartment and interact with the hub.

The problem with this architecture is that a fraudulent service could try to maximize

its profits by accepting a payment from a user but not returning a proof of payment to him or notifying the smart hub of the reservation. Or it could accept two transactions from different users for the same dates, and purposefully reject confirmation requests from the hub for one of them. The problem could even be that the service itself is actually not fraudulent, but a malicious system administrator is behind the attack and is manipulating the system. Either way, the affected user would have difficulty proving he payed for the apartment if records were logged only on the providers side – the Byzantine party in this situation. And even though our example is based on a scenario where the most likely party to behave fraudulently or erroneously would be the provider, in other scenarios any other entity could be the Byzantine one, including the user or the smart hub.

Hence, what we propose is a ledger between these parties to provably ensure which transactions took place and when, by using a blockchain to represent these parties and broker transactions for them, as illustrated in Figure 3.1.

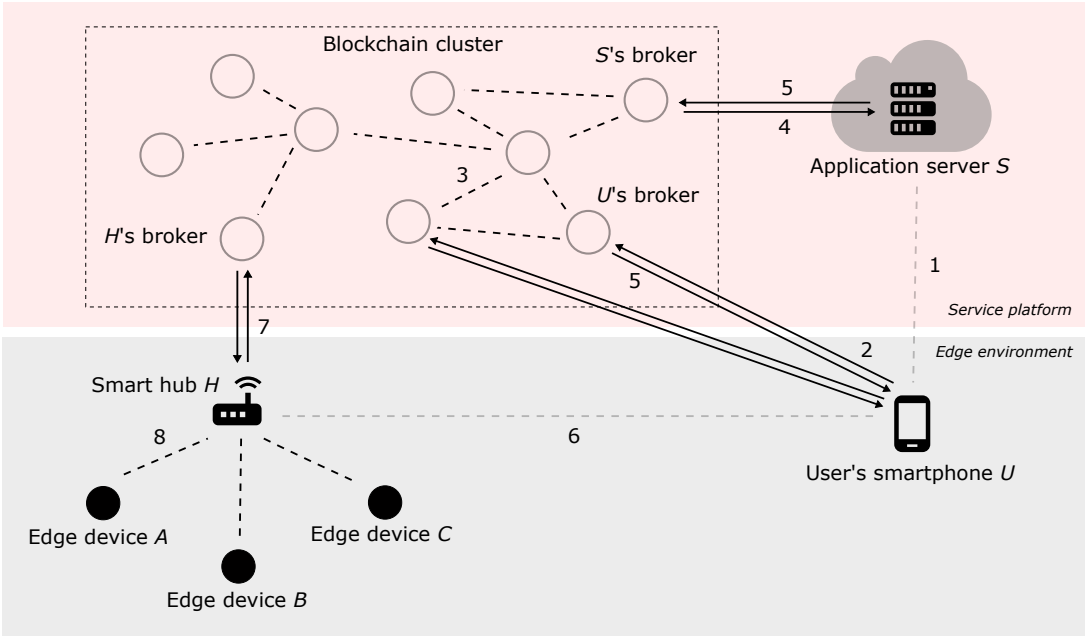

Figure 3.1: Application scenario of the intermediary ledger

In this new architecture, a user, represented here by its smartphone U, communicates with a rental service application server S (1) to obtain information, price, and terms and conditions of the rental. S returns this information along with an identifier for a contract deployed on the blockchain, which U has to comply and interact with. This contract shall specify terms and conditions of the rental process and of the transactions to be executed within the context of the rental. In this scenario, the contract was previously deployed on the blockchain by the provider. U emits a transaction to the blockchain with the intent to rent the apartment, represented by the smart hub H. The transaction is encrypted and signed by U, and sent to a set of nodes (2) specified on the contract which validate the transaction according to the contracts properties, and if successful, sign the transaction

as a way of endorsing it. Then, the signed transaction is propagated throughout network nodes (3), registered to the blockchain and the contract it is associated to regulates the succeeding workflow steps. The transaction result eventually reaches S (4), which can query the its known broker nodes and decrypt the transaction (for now, assume broker nodes to be simply blockchain nodes in direct communication with a given party). If the transaction checks out on S's off-chain validations, S can then reply to U that it has successfully received the transaction via the blockchain cluster (5), in a similar process to the first transaction, which U would then be able to query. When the user reaches the apartment, he communicates with H for it to open the door (6). H will then verify if a previous transaction from U exists within the blockchain representing the user's payment of the rental (7), and validates its information according to the contract specifications. In the end, H commands A (the door) to open (8), and the user gets access to the apartment he payed for.

The potential behind this ledger is that any of the participating parties can consult its brokers and audit any transaction that has happened in the past, preventing fraudulent behaviour and ensuring non-repudiation as transactions are witnessed and endorsed by several peers in the blockchain. It presents itself as an intermediary system and can be integrated with today's popular cloud-first applications for added robustness.

3.2 System Model

Before we burrow into detailing our system model, let us focus on formalizing the participating parties of our system and defining an entity model capable of mapping their relationships accordingly. Afterwards, we will describe the foreseen interactions between these entities in our system model, define base and extended models for such interactions, define system model requirements according to the needs of these entities and the needs of the interaction flow, and finally detail our devised architecture and its components.

3.2.1 Entities

A *Thing*. A device or equipment, equipped with computing and network connectivity hardware, capable of executing simple processing tasks and communicate with other devices and computers via some network. Also referred to as a smart thing, a smart device or an IoT device (e.g. a sensor capable of receiving commands and sending alerts through the Internet or some other kind of network).

The Service Provider. The party responsible for supplying the means and resources for a given service (e.g. apartment or bike rentals, toll services, digital identity services, etc.). The service provider, or simply provider, is responsible for implementing and deploying a smart contract to the blockchain expressing any applicational rules necessary for providing its service to costumers, and is usually instantiated by an application server (or similar) sited on a cloud-centric infrastructure.

The User. The actual user interacting with the system, generally through a smartphone or a similar device. As such, in our system model we may sometimes represent the actual user as its representative medium – the smartphone. The user is the consumer of a service provided by the service provider and information transacted between the user and the provider is recorded on the blockchain.

The Smart Hub. A piece of hardware deployed on the edge capable of executing some resource-intensive tasks that are not suitable to be executed on edge devices with lesser computing capacity, such as smart things. Smart hubs can aggregate data on the edge and perform some sort of computation and even cryptographic services before sending information to cloud services. Smart hubs may be either hardware owned by users (e.g. a smart home hub interconnecting TV, lights, loudspeakers, etc.) or owned by service providers for the purpose of providing specialized services (e.g. a smart hub controlling a bicycle rack platform on a street for magnetically unlocking bikes upon user payment via their smartphone). In the first case, it is possible that users may be able to install several applications and custom software into their smart hubs. Either way, the smart hub presents itself as an intermediary between the user and the provider and any edge devices the hub may need to control and receive information from. It acts as a proxy agent for storing and reading information from a persistency layer. In our system model, this persistency layer is the blockchain.

We must notice that in a generic perspective, the Smart Hub is seen as an intelligent gateway that can be designed to be a pluggable computing software/hardware appliance, where different modules can be installed for running on top of an hardened OS. A possible idea is to look at such modules as virtualized and isolated pluggable software containers, dedicated to their specific functions (e.g. data aggregation and traffic filtering functions using, for instance, tainted-data analysis mechanisms to avoid undesirable data exfiltration/infiltration, inter-networking firewall functions, cryptographic processing functions or other application-specific modules for particular IoT devices).

Another kind of modules would be related with the provision of multiple protocols, suitable for serving specific and distinct IoT environments. This would allow addressing different subsets of IoT standards, from wired or wireless network infrastructure identification and addressing levels, transport-level, discovery and data-dissemination protocols, to request/response under remote-calling operation environments, and data-semantic and representation protocols, including possible protocol-conversion features [48]. In a complete design and implementation of such smart hubs, those protocols can be addressed in the context of multi-layer frameworks, targeted for different IoT markets, devices, and applications. However, we are more interested in regarding smart hubs, for implementation purposes, as possible low-cost software/hardware appliances capable of acting as gateway services to the blockchain supporting local-operation REST-based proxy-service provisioning for natively TCP/IP-supported IoT devices, and CoAP-based REST-style invocations [66].

The Blockchain. The composition of all services, hardware and software which compose a blockchain. Generally, a blockchain is a cluster of server nodes communicating between themselves through a peer-to-peer network holding local copies of a ledger data structure where transactions are recorded. The blockchain presents itself as the backbone of our system model and is the main storage layer of the system where all participating entities read/write from, either directly or indirectly.

3.2.2 Interactions

Base Model. We start with a base model for what we foresee as the most adequate scenario for establishing a blockchain and contract-based service in an IoT-based environment. For our base model, illustrated in Figure 3.2, we consider that users, through their smartphones or another type of mediums, contact the blockchain intermediated by a smart hub. This type of model better fits the IoT environment of smart homes, where there is typically a hub device responsible for controlling and obtaining information from user-owned assets (e.g. an ISP box controlling TV, telephone and acting as a Internet switch, or a *smarter* hub controlling lighting, stereo, heaters, etc.). We also consider that the service providers only job is to deploy and manage smart contracts operating directly with the blockchain. Note that we consider the blockchain not to be an asset of the service provider, albeit this could be a possible situation in a real scenario (imagine it as being a consortium between several providers with shared cloud server farms).

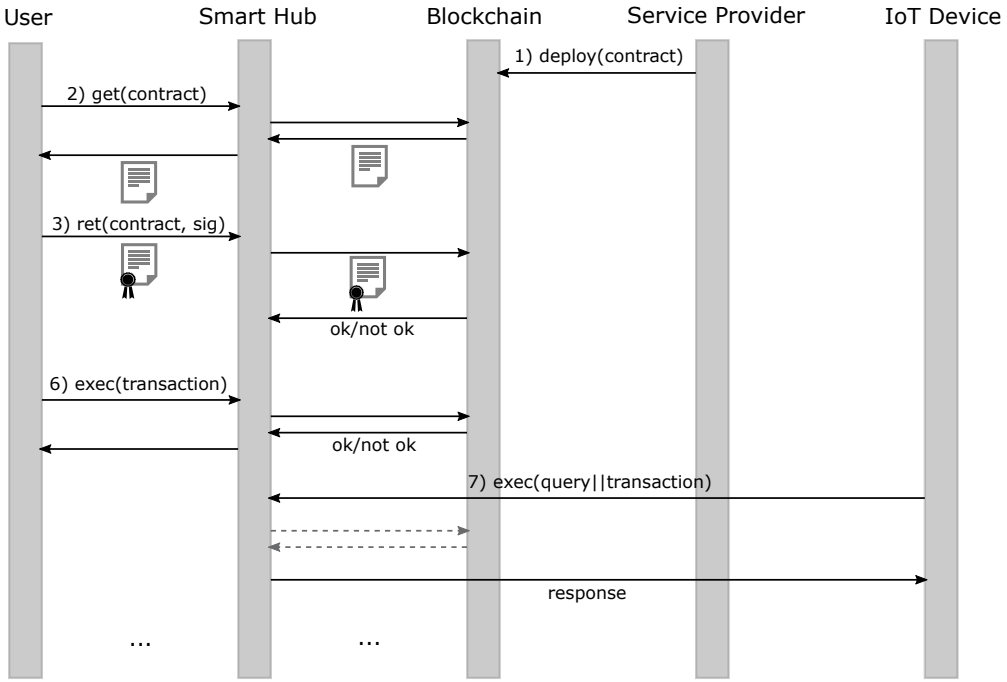

Figure 3.2: Base interaction model

Following on Figure 3.2, the interaction flow of the base model is as follows:

1. The Service Provider deploys a smart contract for a given service, stating terms and conditions of the service, of the transactions to be executed over the contract and how they are digitally signed and verified by nodes in the service;

2. The User, after outside communication with the provider which allowed him to obtain an identifier for a given contract, requests the contract to the Smart Hub. The Smart Hub then either has the contract in cache (from a previous interaction) or requests it from the Blockchain. In the end, the smart contract is returned to the user, which allows him to view and agree with terms and conditions.

3. The User agrees with the contract, digitally signs it, and forwards it to the Smart Hub, which in turn may verify the signature and forward it to the Blockchain. The Blockchain can then either confirm the operation or not (it could fail if the contract was already signed or if there was some application error triggering an invalid state). This process of registering the contract to the Blockchain means that the User has accepted the contract and this action is irrevocable and non-repudiable.

4. The User then executes transactions over the contract. Transaction requests are forwarded to the Smart Hub, which in turn, redirects them to the Blockchain (possibly in a format understandable to the Blockchain), and returns results back to the User.

5. IoT Devices (or Things) linked with the Smart Hub can perform transaction requests to the Smart Hub as well (e.g. a temperature sensor recording Celsius to the ledger, thereby sending a transaction for each temperature reading) or execute queries over data on the ledger (e.g. a lightbulb checking if it should turn on/off). In this illustration, we consider that IoT devices would obtain information from the Blockchain by polling the Smart Hub, but a Smart Hub could also easily implement an event-based approach that would send information to trigger IoT devices.

Extended Model. Now, let us describe a more complex scenario where users also can directly contact the blockchain independently of using a Smart Hub. In this model, the user's medium, the smartphone or other device, has all the needed information and applications to contact the blockchain directly. Note that this feature is not mutually exclusive of the previous model. At home, a user could still use his smart hub to contact the blockchain for a contract regulating smart home services. Outside, he could rent a bicycle via an automated bicycle rack that would require him to contact a smart hub belonging to the service provider of bicycle rents. The advantage of this feature is for situations such as the one in our application scenario – an apartment renting service. It would be illogical that for renting an apartment the user would have to contact a hub to do so. It seems impractical that he would contact his home hub, the rental apartment's hub, or even some kind of open community hub just to perform the actual rent when he could do it himself directly, as modern applications allow you to do today. Thus, the ability to interact with the blockchain directly is vital to achieve a pervasive system.

Another important feature of this extended model, omitted from the base model above, is the ability of service providers remotely configuring smart hubs to some degree, something which can be expected in real case scenarios, especially in cases of hubs that are placed at users homes but are actually owned by the provider for which the user rents the equipment for a fee, or in cases where the service provider has to update and administrate hubs in public areas, such as the hubs on bicycle racks example. These configurations fall under the scope of updating firmware, configuring the interaction procedures with the blockchain, adding new functionalities or deploying contracts directly to the hub.

The extended model can visualized on Figure 3.3.

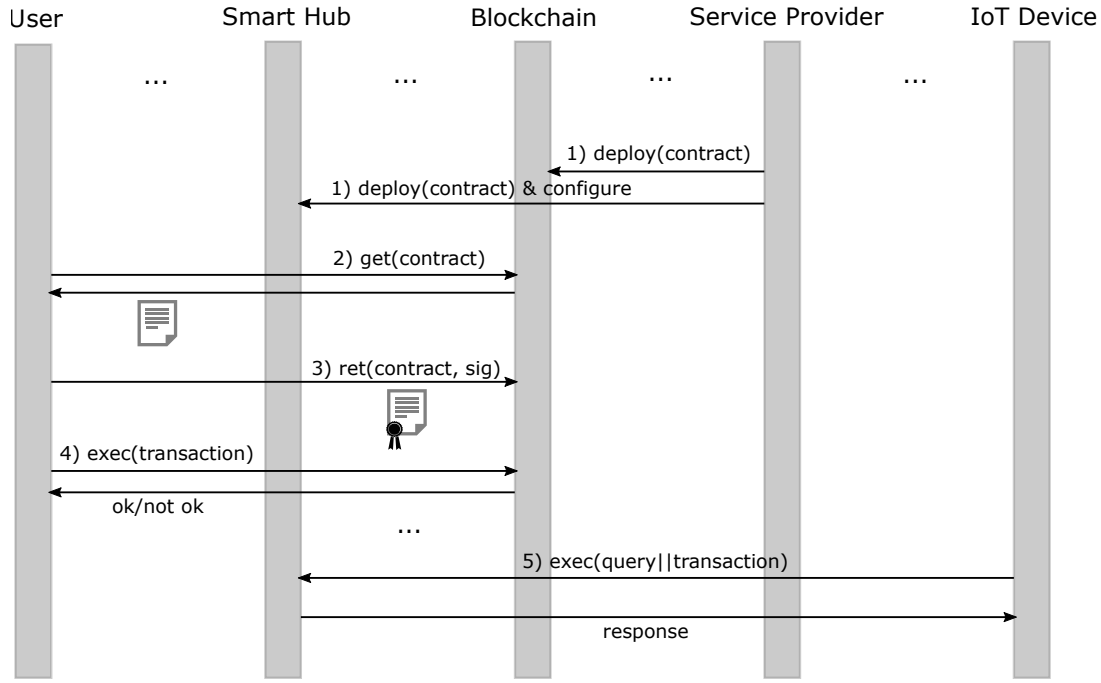

Figure 3.3: Extended interaction model

In this case, the interaction flow is as follows:

1. The Service Provider deploys a smart contract on the Blockchain in a fashion similar to the previous model. However, it can also remotely deploy the contract directly to a Smart Hub and perform some miscellaneous configurations. Imagine the Smart Hub is owned by the provider and is situated inside a apartment up for rental. Additionally, imagine that hub was configured to only use the most recently deployed contract and that all IoT Devices queries/transactions to the Smart Hub shall be routed to that contract's rules and functions.

2-4. The User, which could be on foot and outside, directly obtains a contract from the Blockchain to a service he previously got information from a Service Provider. Agreeing with the terms of the contract, he signs and returns it to the blockchain, and then performs some transactions as he sees fit, all without depending on a

Smart Hub. Let us imagine he accepted a contract stating rental conditions and afterwards executed the payment for the rent.

5. An IoT Device that most likely has no idea of what is a smart contract, contacts a nearby Smart Hub, which has been updated and configured by the Service Provider. Imagine it to be a heater intended at heating the apartment 15 minutes before a guest arrives that is trying to get the current time from the hub and know when does a guest arrive. The Smart Hub can then check its contained smart contract used to regulate rentals, query the blockchain for signed contracts to identify future guests and their expected time of arrival, and reply to the IoT Device. Overall, the User was independent of communication with the Smart Hub for this process, and the Hub was able to respond directly to the needs of IoT Devices through the configuration made by the Service Provider.

3.2.3 Requirements

While the scenario described in Section 3.1 is merely illustrative, the previous sections exemplify the ability of such a system to apply to different application-specific business needs. However, in order to present a generic system model, useful for any kind of IoT application, we need to derive some requirements on what we intend to achieve of our system model. As such, our system model should[1]:

R0 Allow for replicated tamper-resistant and cryptographically-secured persistent storage of data and information.

R1 Be scalable in terms of the number of applications concurrently deployed on the system, without major performance or functionality degradation with an increased number of service provider applications.

R2 Be scalable in terms of the number of users concurrently reading and writing information on the system, without major performance or functionality degradation with an increased number of users, independently of the number of existing applications.

R3 Support BFT on consensus, data persistency, read/write operations, transaction authentication, signing and verification processes, enabling for trustable and resilient ledgering, where erroneous behaviour or deliberate attacks targeting the system as a whole, the operation of IoT environments, or users, does not succeed in compromising the system and endangering its users.

R4 Be decentralized in terms of trust, allowing data to be cryptographically signed and verified by multiple entities in a fault-tolerant fashion, and allow the system to be deployed in an environment that does not promote vendor lock-in and infrastructural or organization centralization and control.

[1] We will refer to each requirement by its respective identifier (R0, R1, ..., Rn) on the following sections.

R5 Ensure clear identification of the participating entities in the network and ensure non-repudiation of operations to its persistency mechanism by any involved party.

R6 Allow for the auditability of any operation executed in the past by any participating party independently and allow the traceability of that operation to a transaction, to a contract and to an entity.

R7 Be able to present itself as a generic platform for several customizable application contexts, each with different business needs.

R8 Allow service providers to express a form of extended smart contracts that state a set of terms and conditions to be accepted by users, and be capable of interpreting such contract at both application-level (business-specific functions, such as price of a rental) and system-level (signing policies to use, smart contract identifier on the blockchain, consensus mechanism to use, and other properties), acting accordingly.

R9 Allow complete execution of the protocol without degrading IoT devices performance or overall experience, despite of resource limitations, and without degrading the protocol's efficiency and performance by the presence of resource-limited devices in the system.

R10 Enable the scalability of cloud providers (given the exponential ever-growing numbers of the IoT) by providing a model capable of aggregating IoT devices and data on the edge and being deployed heterogeneously across providers, thereby also avoiding centralization by vendor lock-in.

3.3 Reference Architecture

Our proposed system poses as a robust mediator between several participating devices enabling cryptographically-secured auditability and non-repudiation of transactions by any of the involved parties. To achieve this in terms of architecture, we devised a middleware set of services over a generic layer of base blockchain services we refer to as extended blockchain services. Then, we devised a higher-level applicational interface – an API – composed by several sub-components intended at being deployed within a smart hub for it to be able to perform the expected interactions of our system model. Together, these three layers, illustrated on Figure 3.4, make up a fully fledged system capable of responding to the needs of external parties: users, their devices, *things*, and service providers. As our system model is essentially a distributed system over a peer-to-peer network, this architecture is to be viewed from the perspective of a single smart hub and a single node present on a blockchain network in communication with the hub, with each *sibling* blockchain node and sharing the same set of components.

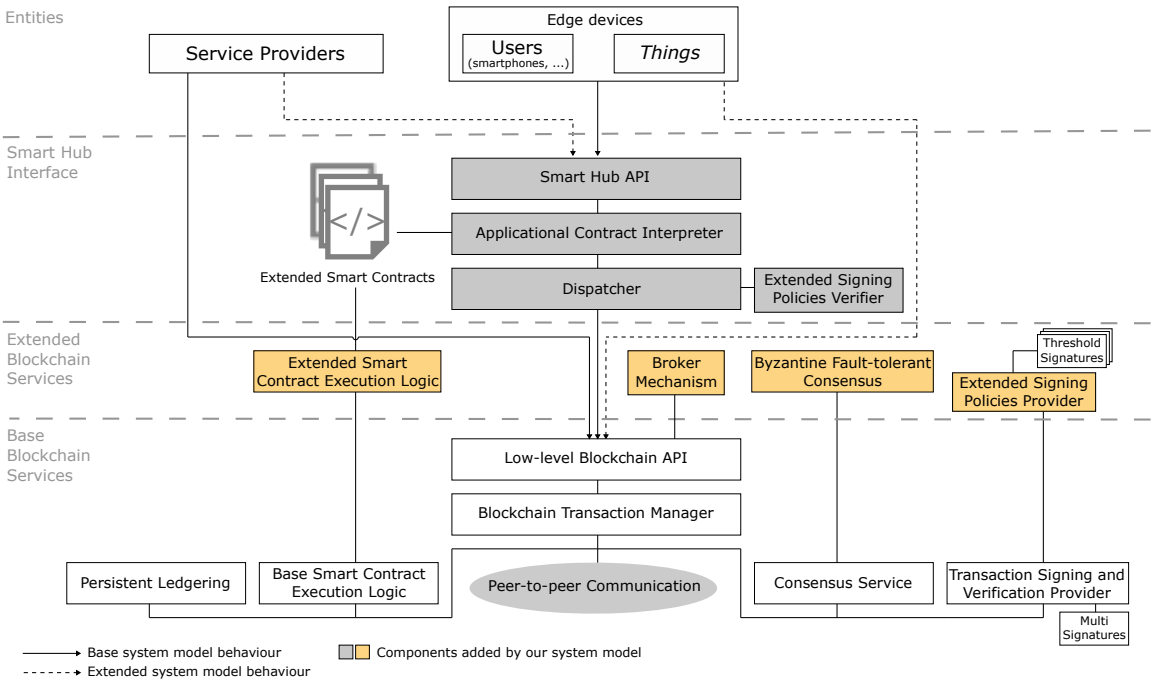

Figure 3.4: Architectural view of the system

Base Blockchain Services. Our system model intends to be independent of a single blockchain platform. Yet, we have to consider the generic functionalities given by virtually every blockchain platform to clearly identify which blockchain components we may need to extend, add or use as-is, according to the requirements in Subsection 3.2.3. This system model is oriented to blockchain platforms harnessing consortium-like consensus, rather than proof-of-work. In general, such platforms expose a low-level API on each node for executing read or write operations over a ledger. This API interprets requests, processes them, and passes them on to a core orchestration component we call a transaction manager. The transaction manager has access to a wide array of components to execute the operation required by the incoming request, including components for system administration and smart contract deployment by service providers. Standard blockchain transaction flow for a write operation usually requires that the incoming request is transformed into a blockchain transaction and is verified and executed against a smart contract engine/execution logic, without necessarily persisting the result of the execution of the contract on the ledger. These execution and verification processes can be done by multiple nodes, requiring the transaction manager to propagate the transaction to the peer-to-peer network, or by a single node, depending on the platform at hand. The transaction is then, generally, signed by the nodes that are verifying it and a consensus algorithm is executed. Upon reaching consensus, signed transactions are propagated across the network, executed against the smart contract engine and persisted to the replicated ledger after successful verification of its signatures by peer nodes. This layer is responsible for ensuring requirements **R0**, **R6** and **R7**, and most blockchain platforms already provide some form of node identification through public-key cryptography, ensuring **R5**.

Extended Blockchain Services. At the next level of our architecture sits the extended blockchain services layer. This layer is responsible for enriching the base blockchain services layer with mechanisms for further decentralizing trust in transaction endorsement and signing processes, for providing a consensus service capable of tolerating Byzantine behaviour, for implementing, together with the base blockchain services layer, the broker mechanism, and finally, for providing an engine with enriched smart contract execution logic to execute what we define as extended smart contracts – contracts specifying not only the business logic of different applications, but also system properties that nodes should be able interpret and execute accordingly to conclude the required operation (e.g. imagine a transaction that has to be signed using decentralized trustability assumptions from five trusted and known nodes listed on the contract). The concept of a broker mechanism is fairly simple and was introduced in the application scenario in Section 3.1. The idea behind it is to use a set of blockchain server nodes to represent entities external to the blockchain, which are, in essence, clients of the service. Thus, a broker is a blockchain node that is known to an entity to which it requests blockchain operations to be performed (queries, transaction proposals, etc.). A node that is not acting upon an entity's request is not a broker node and is simply supporting the peer-to-peer network. This approach ensures **R9** as all resource-intensive tasks related to operating the blockchain are offloaded from the actual edge devices to resource-wise capable server nodes that broker operations for them. It allows an aggregation of smart hubs and edge devices (the latter only on the extended model) to a single broker, enabling for a more scalable approach to the system that doesn't require an exceptionally high number of nodes to manage within the ledgering system, therefore aiding in complying with **R1** and **R2**. However, ensuring these two requirements is dependant on the base blockchain platform to be used, as the inner operation of the blockchain protocol heavily impacts scalability. As such, the comparison of blockchain platforms in Section 2.4 is a significant factor to be taken into account. This layer is also the basis for compliance with requirements **R3**, **R4**, **R8**.

Smart Hub Interface. At the final level of the architecture the smart hub interface can be found. The purpose of this layer is to provide a high-level interface for edge devices to attach to in order to make use of the decentralized ledgering system. This layer is what allows the aggregation of data, devices and users and computation on the edge, ensuring **R10**, as it allows a far more scalable alternative to cloud-first architectures in which all devices communicate directly with a cloud service. This layer is responsible for holding and partially interpreting the properties of extended smart contracts built by service providers (partially because the remaining part of the smart contract logic is to be executed in the lower layers). Upon successful interpretation of a contract, the smart hub interface makes use of a dispatcher component that orchestrates communication with the blockchain's nodes for the remainder of the protocol. An important distinction to make in our system model is how would this layer operate for the extended interaction model versus the base interaction model. In the base model we restrict communication

between edge devices and the blockchain to be routed through this API, however, for the extended model, it would be necessary that for edge devices to be able to communicate with the blockchain they would have to implement a similar layer of components (contract interpreter, dispatcher module, extended signing policies verifier) as to ensure the same capabilities of a smart hub. Naturally, this would only apply to devices that have a reasonable amount of hardware resources (e.g. smartphones) and not by cheap resource-constricted IoT devices such as temperature sensors and the like that would not be capable of communicating directly with the blockchain. The other feature of the extended model is that the service provider would be able, through some sort of backdoor, to configure smart hubs, as illustrated on the architecture diagram.

Computing environment. It is our intention with the definition of this architecture to achieve a modular system, in which nodes can be heterogeneous in components. This allows the presence of auxiliary nodes on the network, i.e. nodes that may not need to sign transactions and act as *witnesses* of a transaction probably do not need to have an extended signing policies provider installed on them, or nodes that are passive to their environment and only record information to their local ledger probably do not need a smart contract execution logic deployed within them. As for the environment of the system, we intend for it to be deployed throughout different cloud service providers, as to prevent vendor lock-in and conform with **R4** regarding decentralization of trust. Thus, this approach complies with **R10**.

Final remark. With this model we take into account the concerns that most IoT devices aren't able to cope with the weight of the blockchain protocol resource-wise. We also believe that network-wide consensus and global ledger replication are properties that weigh in on the protocol's scalability. In partial detriment of decentralization, a new protocol can be built to be secure, partially-decentralized and performant simultaneously. In addition, some blockchain platforms bring confidentiality mechanisms to transaction processing, which can be an essential property within the scope of IoT applications. One could imagine a network supporting a wide array of different yet related services which remain private while still interacting with each other (booking an hotel room, requesting room service, paying for the hotel's restaurant, etc.), opening the potential for a wide array of applications. Our model is inherently tied to the notions of a permissioned blockchain setup, in which nodes could be managed between a consortium of organizations. Thus, these criteria, along with those in Section 2.6, influence the blockchain platform we use for our base blockchain services layer, presented in Chapter 4.

3.4 Software Architecture Components

In this section, we address the architectural components of the two uppermost layers of our model and specify the blockchain middleware that enables Byzantine fault-tolerance together with decentralized trustability assumptions in blockchain consensus and signature verification processes for our system model. We do not approach components of the

base blockchain platform here as these depend on the blockchain platform being used and on their internal specification.

3.4.1 Smart Hub Interface

Smart Hub API. This is the component that serves as an entry-point for smart hub communication and to the decentralized ledgering system, providing a high-level API for the entities described previously (devices on the edge). Let us define the method stubs of the methods the API is expected to support:

- `Contract getContract(contractId)`: Obtains a contract definition from the blockchain stating all needed information about the contract represented by `contractId`;
- `boolean signContract(contractId, signature)`: Requests to sign a given contract, represented by `contractId`, with a user-provided signature as a form of indicating acceptance of the conditions and definitions of the contract. Returns a confirmation if the signature was committed to the ledger;
- `LedgerData[] query(contractId, operationId [, queryParams])`: Performs a query operation on the ledger over a specified contract, represented by `contractId`, with the possibility of stating query parameters (e.g. find by ID = 2, return all entries previous to a timestamp, etc.). The query operation to execute is identified by the supplied `operationId` (e.g. queryApartments, queryRentals);
- `boolean invoke(contractId, operationId, transaction)`: Invokes an operation, identified by `operationId` (e.g. rentApartment), over a contract, represented by `contractId`, and proposes a transaction to the ledger. Any arguments/parameters needed for the operation to execute are included within the `transaction`. Returns a confirmation if the transaction was committed to the ledger.

The methods `getContract` and `signContract` are to be invoked by system users for obtaining and accepting (signing) contracts deployed to the system by a service provider. On the other hand, `query` and `invoke` are to be used by all edge devices: devices representing consumers of the service, i.e. the actual users, and IoT appliances and things. These methods objective is to allow querying the ledger state and modifying it by proposing new transactions. Access control policies can apply to these methods, and in finer granularity to the smart contract operations they invoke. However, we defer these access control policies to be specified on the smart contract and not on the smart hub or any of its components.

We foresee that this component may be implemented using different protocols such as HTTP, MQTT (Message Queuing Telemetry Transport) [47] or CoAP (Constrained Application Protocol) [66], giving preference to the latter two protocols for interactions with resource-constricted devices as they are far more lightweight than standard HTTP

and a multitude of sensors and other equipment in the market have started adopting them. Communication could be secured with lightweight security protocols such as DTLS [59], far more friendly for the IoT than standard TLS in terms of resource consumption. A smart hub API could even be made to support more than one protocol and could segment users and IoT things into each protocol (e.g. HTTP for users on their smartphones and personal computers, CoAP and MQTT for remote sensors and actuators).

Extended Smart Contracts. The structure of what we define as an extended smart contract is visible in Listing 3.1. Here we can see that a contract holds three essential sections: *i*) a section for the properties of the base blockchain platform; *ii*) a section for the extended system functionalities we intend to add to the base platform; and *iii*) a section for business logic and application-specific properties. The contract also keeps track of data related to it – the transaction log, which provides a collection of the history of transactions executed over this contract, and the records of data stored within the ledger related to the contract and its transactions.

Listing 3.1: Structure of an Extended Smart Contract (ASN.1 notation)

```
ExtendedSmartContractProperties DEFINITIONS ::= BEGIN

    Transaction ::= SEQUENCE {
        Id              OBJECT IDENTIFIER,
        Payload         BIT STRING
    }
    LedgerData ::= SEQUENCE {
        Key             OBJECT IDENTIFIER,
        Value           BIT STRING
    }
    BasePlatformSystemProperties ::= SEQUENCE {
        ContractId      OBJECT IDENTIFIER,
        TransactionLog  SEQUENCE(SIZE(0..999)) OF Transaction,
        LedgerRecords   SEQUENCE(SIZE(0..999)) OF LedgerData,
        ...
    }
    ExtendedSystemProperties ::= SEQUENCE {
        SignatureType   INTEGER(0..3),
        WitnessNodes    SEQUENCE(SIZE(0..10)) OF IA5String,
        ...
    }
    ApplicationProperties ::= SEQUENCE {
        ...
    }
END
```

Of course, a smart contract is not only a set of properties and data, it also provides the functions and logic needed for it to be executed. Overall, a smart contract is a set of stored properties together with triggerable procedures and functions. Again, the internal logic of such functions is divided into three types: *i*) base platform execution logic, which

is the boilerplate code for running the contract over a smart contract engine; *ii*) extended execution logic, which is logic meant to be used for executing protocol operations according to the extended system properties section, such as the number of nodes to sign a transaction and what cryptographic signature types to use; *iii*) and finally application-specific functions, which are to be implemented and defined by service providers before deployment of the contract.

Our intent with this extended smart contract notion is also that there can more than one implementation of smart contracts within the system. The whole specification of a contract with its procedural logic and functions will always lie at the base blockchain services layer, but at the application-level interface and extended blockchain services layers, another *view* of the contract, or a subset of its information, can be used. Congruently, the relevance of the information present in each section of the smart contract depends at which layer it is read, as some properties will influence business data while others will alter the overall execution of the blockchain protocol. For example, a property that indicates a given set of nodes to verify and sign some transaction will influence an application to dispatch requests only to those nodes, while a property indicating to increase a counter variable in the contract will presumably only alter the application data.

The power of these contracts is their ability to present themselves a generic boilerplate for mapping complex applications to smart contract logic. One can imagine a contract for regulating transactions for electronic payments, in either a peer-to-peer fashion where no centralized entity exists (e.g. the Bitcoin protocol) or a more structured scenario where transactions are made between a specific set of entities, where each recipient can audit the contents of the transaction independently without needing to know the whole contents of the transaction, i.e. the recipient may be able to verify only a section of the transaction relevant to its operation with every other section obfuscated to him, thus retaining privacy control over the contents of the transaction. Advanced control flows of this transaction can be implemented in the contract, such as requiring signatures from specific entities in order for it to succeed. This is an example application of the SET (Secure Electronic Transactions) protocol [68] implemented in debit/credit card payment schemes.

Applicational Contract Interpreter. This is the component responsible for partially interpreting extended smart contracts at the smart hub interface layer. This interpretation is executed over a *view* of a given smart contract which can be loaded directly from the blockchain by its identifier. The *view* of a contract is simply a variation of the format of the original smart contract, the latter being kept at the blockchain service layers, in which only certain properties relevant for the Smart Hub Interface layer are retained (e.g. smart contract functions are not interpreted or executed at this layer, and thus, there is no need to pull them here). As such, this component implements a local persistent storage for the retaining *views* of contracts. The need for this interpreter lies in the next component – the dispatcher – which needs to know a certain set of properties before forwarding requests to the appropriate nodes in the network, such as which nodes to contact, which signature

policy should they use, which consensus service (BFT or fail-stop), and so on.

Dispatcher. The dispatcher component can be seen as the component that orchestrates all communication with blockchain nodes, via their low-level API, in order to achieve read or write operations over the ledger. To do so, it requires information to be passed on to it by the applicational contract interpreter, so it can proceed to emit requests in conformance with a given contract's rules. Thus, the dispatcher has to be able to request nodes to verify, sign, and commit transactions to the ledger.

3.4.2 Extended Blockchain Services

Extended Contract Execution Logic. As stated previously, our notion of extended smart contracts requires contract's actions to be executed at different stages of system architecture. Thus, this component is meant for extending the base contract execution logic of the base blockchain platform with the capability of executing actions that depend on the extended blockchain services or smart hub interface layers and cannot be directly handled at a lower level. This execution logic will be embedded in the blockchain platform as to call procedures from the extended signing policies provider for signing and verifying transactions whose contract requires some decentralized and fault-tolerant transaction endorsement scheme, to require a transaction to be executed over the BFT consensus service, or to contact some sort of external service that does not readily exist within the decentralized ledgering system.

Broker Mechanism. When a new entity joins the system, a user, a smart hub or another device, it is assigned to a predetermined set of brokers (in the base model this will only occur with the smart hub). These nodes can be seen as bootstrap nodes on the blockchain network, as they are the nodes that will initially respond to the external entities requests and act on their behalf. Note that this does not restrict the joining entity to communicating only to these nodes. Service providers can also have their own brokers, which are likely to be their own servers, to represent themselves within the system, from which they manage their presence on the service and their contracts. We assume, for a simplified approach, that the entity joining the service has a set of pre-exchanged public-keys and any other cryptographic material with the brokers (e.g. by configuration) in order to be able to communicate with them in a secure fashion. A more complex scenario could be implemented through a dynamic discovery mechanism where an entity polls its known brokers for additional brokers to communicate with. While this mechanism is conceptually located at the extended blockchain services layer, in practice it is expected to be implemented in part by the base blockchain services, for accepting external entities communications in a client-server architecture and for maintaining a record of which entities do they represent, and in other part by the entities themselves, which will have to perform requests to blockchain server nodes and interact with bootstrap broker nodes and use them to find any other needed peer nodes in the blockchain network. One thing to

note is that if the base blockchain platform does not execute global ledger replication, each broker will need to be identified as a representative proxy for a given external entity as to replicate any needed data to that external entity on its brokers. This broker mechanism, which links an entity to a set of nodes, together with the internal identification of nodes already supported by the base blockchain platform, ensures **R5**.

Byzantine Fault-Tolerant Consensus. As described in Section 2.4, some blockchain platforms using consortium-based consensus operate under the premise of supporting crash fault-tolerant behaviour only. While there is in fact a certain number of platforms that provide Byzantine fault-tolerance, this is not always the case. We intend for our platform to endure any malicious or faulty behaviour to its consensus mechanism that may corrupt how nodes view the replicated information on the ledger.

The component described here is essentially an extension of existing consensus mechanisms in the base blockchain services layer that may provide only crash fault-tolerance guarantees. It will be based on a PBFT state machine replication algorithm [15] and will be responsible for receiving proposed transactions from blockchain nodes, grouping them into a single unique block to inserted into the blockchain, and distributing this set of ordered information back to the nodes for them to append to their local ledger. Our intended architecture for this consensus service is further described in the next section.

Extended Signing Policies Provider. Every node and broker on the blockchain network will leverage some type of signature scheme for the purpose of endorsing transactions. In general, transaction signing and validation flows within blockchain systems employ some variation of multi-signature schemes. We intend to override this basic policy for enabling other schemes that further enable decentralized trust and fault-tolerance over standard blockchain transaction flows. By doing so, we increase robustness by allowing transactions to be signed by multiple entities in a fault-tolerant fashion, while retaining the ability to compare several signature schemes regarding performance and robustness. Accordingly, this component's objective is to supply the blockchain platform with a set of primitives and software libraries for using multiple transaction signing and verification schemes (or policies).

Our system model shall harness the following policies, while allowing for the extensibility of adding new ones:

- *Multi-signatures*: Standard signing and verification policy we assume to be present in the base blockchain services layer, where a multitude of nodes sign a single transaction and a transaction is committed along with a given set of signatures, each one uniquely representing a single node and verifiable by the public-key corresponding to that node. Generally, blockchain platforms resort either to RSA or ECDSA signature schemes.

- *Threshold signatures*: Signing policy added by our extended signing policies provider component, based on the threshold signature schemes described in Section 2.5,

where nodes share a group public-key and sign using shares of a private key. The result is a single signature and a single public-key to verify that same signature. We intend to focus on RSA threshold signature schemes.

3.4.3 BFT Middleware for Decentralized Transaction Flows

BFT Consensus for Blockchains. We propose a model for harnessing a consortium off-chain consensus service that is tolerant to Byzantine behaviour and can be applied to permissioned blockchain transaction flows. We deviate from fully decentralized consensus algorithms that rely on proof-of-work, as these are computationally costly and exert strong latency conditions upon transaction confirmation, and upgrade upon fail-stop consortium models that assume no faults shall occur that do not fall within the spectrum of system crashes.

PBFT [15] and Tendermint [71] are two examples that have been harnessed for blockchain systems requiring BFT consensus for converging on ledger state.

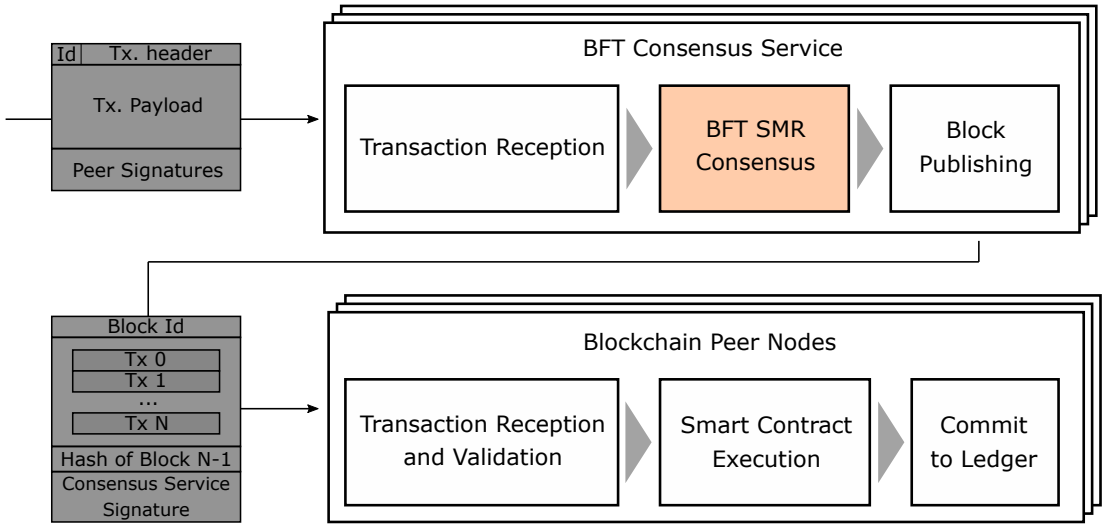

Figure 3.5: BFT consensus service and ledger block publishing

To achieve such a model, we impose a few requirements on the BFT consensus service. First, the underlying SMR protocol has to ensure atomic broadcast and hold the consensus properties of safety and liveness in an asynchronous environment, requiring $3f + 1$ replicas to be able to tolerate f faults. Secondly, after reaching agreement on the total order of received transactions, the service shall maintain a reference to the last verified published block of the ledger, collect all received transactions up to a given limit of maximum block size or maximum transactions to aggregate in a single block, and produce a new block with the transactions and an hash of the previously published block, digitally signing it before relaying it across blockchain peers. Peers receiving the new block will then verify that the signature belongs to a trusted consensus service, check that it is in fact the next block in the ledger sequence, and verify transactions one-by-one: i) by checking

the appended peer signatures of each transaction; and then *ii*) by executing the transaction payload against a smart contract engine. If every validation succeeds, the block is appended to ledger. If a validation fails, either a single transaction or the whole block may be discarded and peers await for the next block. Figure 3.5 outlines this process.

Threshold Cryptography for Decentralizing Blockchain Transactions. As mentioned before, we intend to complement transaction signature and verification processes with threshold cryptography. Such a scheme can prove itself quite useful for removing single points of failure in standard signature processes, in which a single signature belongs to a single peer, and in which a failure to sign a transaction by a single peer, independently of how many signatures may have already been collected, may result in forfeiting the transaction. To accomplish this, we have to define a framework on which a blockchain architecture can base itself to benefit from a threshold signature scheme. We will describe this framework in a manner similar to Stathakopoulou and Cachin [69], as their approach to explaining this is quite clear, but some of these concepts have been defined and scrutinized in the past and are commonly known in the field. Let us first define the method stub of what we expect from such library:

- `<GroupKey, KeyShare[]> deal(n, k, mod)`: Takes the number of participants `n`, the minimum quorum size `k` and the RSA key modulus size `mod` as input in order to generate all needed cryptographic material to be used for an RSA threshold signature scheme, namely: the public group key to verify a threshold signature against, and an array of key shares, each private to a single peer;
- `SignatureShare sign(m, ks)`: Given a private key share `ks` and a message `m`, produces a signature share over `m`;
- `boolean verify(m, sigs, gk, k)`: Given a message `m`, a set of signature shares `sigs` and a group key `gk`, verifies the validity of the signature over `m` against the respective public group key. Internally, it assembles a composite signature from the given signature shares and the minimum quorum size `k`. Depending on the implementation it may also internally verify the validity of signature shares individually. The verify method fails if the threshold of valid signature shares is not met or if the composite signature itself is not valid.

Given this method stub, it is clear that nodes on the blockchain will resort to the methods `sign` and `verify` for the transaction flows within the blockchain through the extended signing policies provider. As method `deal` outputs the complete set of private key shares to be used by participants in the blockchain network, it is also clear that this scheme requires a trusted dealer to execute this method and securely distribute key shares between peers. For a network that is inherently dynamic, where existing nodes can leave and new nodes can join, such a method would be successively invoked and cryptographic material continuously distributed by a dealer, in order to keep peers updated with their

respective key shares and group key. As discussed in our related work, an interactive key generation scheme could also be achieved, but it would require a higher overhead for blockchain nodes whenever the network topology changed. We do not discard this approach, but we leave it for future work if one wishes to study such a scheme.

The signature verification algorithm, triggered by invoking `verify` is perhaps the most important method we have to address here in terms of computational cost. Threshold signatures are costly to verify, and any robust threshold signature must tolerate the presence of f corrupted signature shares as long as the total set of signature shares n is superior in number to the threshold k. In other words, a signature has to validate if $f \leq n - k$. Accordingly, threshold signature verification algorithms can be classified according to the expectancy of corrupted signature shares fed to them and to their expected computational cost, in pessimistic and optimistic variants of the algorithm, Algorithms 1 and 2, respectively. The notion of the algorithm being pessimistic or optimistic refers to the task of assembling signature shares into a single signature to be verified against the public group key.

Assume that to support these algorithms, four internal auxiliary methods exist: `buildSignature` whose purpose is to create a composite signature when fed a set of signature shares, `nextCombination` that is able generate a combination of signature shares of size k given a set of size $n \geq k$, `verifySignatureShare` which may be used to verify an individual signature share, and `verifyCompositeSignature` which verifies a signature over a given message similarly to the traditional RSA verification algorithm. Further assume that `nextCombination` behaves in a static way and that each call to this function will always generate a single combination in a correct sequence.

Algorithm 1 Pessimistic Threshold Signature Verification

```
1:  function verifyPessimistic(m, sigs, gk, k)
2:      valid ← []
3:      i ← 0
4:      for s ← sigs do
5:          if verifySignatureShare(s) then
6:              valid[i] ← s
7:              i ← i + 1
8:      if len(valid) ≥ k then
9:          compositeSig ← buildSignature(valid)
10:         return verifyCompositeSignature(compositeSig, m, gk)
11:     return false
```

The pessimistic approach assumes that in a given set of signature shares, some of them are corrupted, and thus verifies each and every one of them until it reaches k signature shares needed to verify a composite signature, discarding corrupted ones. Naturally, this has a heavy cost in the execution time of the verification function.

The optimistic approach, on the other hand, assumes that the first combination of k

Algorithm 2 Optimistic Threshold Signature Verification

```
1: function verifyOptimistic(m, sigs, gk, k)
2:     c ← []
3:     repeat
4:         c ← nextCombination(k, sigs)
5:         if len(c) > 0 then
6:             compositeSig ← buildSignature(c)
7:             return verifyCompositeSignature(compositeSig, m, gk)
8:     until len(c) = 0
9:     return false
```

signature shares it retrieves are valid and proceeds to verify them immediately. If verification fails, the algorithm will then retrace its steps back to generate a new combination and repeat the process, until a valid signature is found or all combinations are exhausted, failing the verification.

These threshold signature verification algorithms are in the category of synchronous algorithms, but they can be made asynchronous with the introduction of a communication channel into their execution. We expect that any of these schemes may be leveraged in a implementation basing itself on our system model.

3.5 Threat Model Considerations

The design of our solution takes some considerations for the definition of a subjacent threat model that we point out here.

External intrusion. Our threat model is focused in the design of the blockchain-based support and middleware provided services in our platform. The external client-side usage of the platform is considered as a trustable environment. Our platform must operate under the assumption that malicious adversaries may want to corrupt the history or logging of verifiable transactions, revert or compromise the process of transaction validation or even compromise data sharing and related logging operations. The back-end blockchain support must be able to avoid this by applying the necessary security guarantees and countermeasures, defending from intrusions that will try to induce such corruptions and incorrect operation. The counter-measures will be imposed by the Byzantine fault-tolerance guarantees of the inherently provided consensus protocol primitives.

Internal attacks. In fact, most blockchains operate under the premise that they will be attacked not only by outside adversaries but also by authorized users executing operations under possible malicious or incorrect behaviours. The expected threats and the degree of trust that the network has in the participating nodes in the global blockchain determine the type of the required consensus algorithms that they use to settle their ledger. As in other existent blockchain platforms, we take as reference a threat model considering

a very high degree of possible threats, that leads to the need of strong consensus mechanisms based on BFT assumptions and leveraged by intrusion-tolerant state-machine replicated processing. However we assume that the adversary never compromises the minimum number of correct nodes required to support the safety and liveness conditions of the required BFT consensus protocol. In this we include the provisioning of the base properties of strong consensus guarantees at each node level running and endpoint of the consensus protocol in our blockchain-based platform, namely; validity, agreement, integrity and termination [18].

Exploit of communication channels. We consider that the network (supporting the interactions between users and nodes) can be controlled by potential adversaries, starting by defining in the adversary model all the identified threats against user-to-node or node-to-node communications, as stated in the attack typology and reference of the OSI X.800 Security Framework [2]. Thus, our interactions have to be supported by a trustable establishment of user-to-node and node-to-node secure channels, guaranteed by mutually authenticated handshakes and TLS 1.3 support, enhanced by the possible complementary verification of resilient threshold digital signatures of all operations related to the verification of smart-contract rules and blockchain transactions.

Trust computing base. Considering the system model and architecture, we consider that processing capabilities installed on the edge and intermediated by local smart-hubs are in the trust computing base. Thus, we consider in our design principles that the intermediary smart hubs are trusted (including by local hardware, firmware and software), and the specific applications running in user-controlled computing devices are also considered in the trust computing base. We do not consider attacks against the availability of computational and communicational resources, namely those frameworked as DoS or DDoS attacks typology. We also do not consider in our primary assumptions routing attacks in the peer-to-peer intercommunication environment, namely attacks such as: blackholes, sinkholes, wormholes or sybil attacks causing the loss or exhaustion of processing and communication capabilities.

3.6 Runtime Behaviour

We will now proceed to describe the planned runtime behaviour of our system and its components in two types of operations: *i*) a write operation that always involves proposing a transaction to the blockchain; and *ii*) a read operation, that simply reads whatever is on the ledger, depending on what was specified on a smart contract for its execution. These two operations are the basis for every other operation over the blockchain (even operations for obtaining a contract from the blockchain or for signing it). We will restrict ourselves to the base model of interaction, as the extended model can be inferred

[2]OSI X.800 Security Framework specification: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.1145-201705-I!!PDF-E&type=items

from this one (as it would simply require for edge devices to implement a similar set of components to that of a smart hub in order to communicate with blockchain nodes).

Proposing a transaction. Figure 3.6 illustrates a flow diagram of the execution of the write protocol. Consider this execution to be done after a user has already accepted a given service provider contract.

1. The user, using his smartphone, initiates a transaction with a nearby smart hub for a given service, referring to a smart contract existing in the blockchain by its ID;
2. The hub receives the transaction via its API and forwards it to the application contract interpreter which will validate the specified contract's properties to be used for the remainder of the execution of the transaction. In this case, the interpreter has the contract cached in local persistency. If that was not the case, it would have to fetch it from the blockchain by means of a read operation to its broker nodes;
3. After the contract properties are read, the transaction is passed on to the dispatcher, which will orchestrate the transaction execution over the blockchain peer nodes;
4. The dispatcher invokes the low-level API of the blockchain platform, proposing the transaction and requesting blockchain nodes to verify it and sign it. The nodes to propose to, which are the signing (or *witness*) nodes, are specified on the contract;

5-8. The nodes validate the transaction by executing it on the specified contract over the extended smart contract engine. This engine is designed to execute dynamically according to the contract properties;

9-11. After successful validation of the contract, nodes proceed to sign it, using the policy described on the contract. If the contract specified a different type of transaction signing policy than the supported default policy by the base blockchain platform, peer nodes request a special provider component to sign the transaction for them;

12-13. If nodes correctly verify the transaction, they reply to the hub, which collects the signatures at the dispatcher component. Receiving the set of signatures from replying peers, the hub proceeds to verify them according to the signing policy specified on the contract;

14. After enough signatures are collected and validated, the hub sends the transaction to the consensus service;
15. Consensus is reached and the consensus service propagates the transaction to every peer, including passive peers who did not *witness* the transaction proposal;

16-19. The transaction signatures are verified by the receiving peers and if everything checks out, the result of executing the transaction on the contract, using the smart contract engine and any needed extended logic, is committed to the ledger;

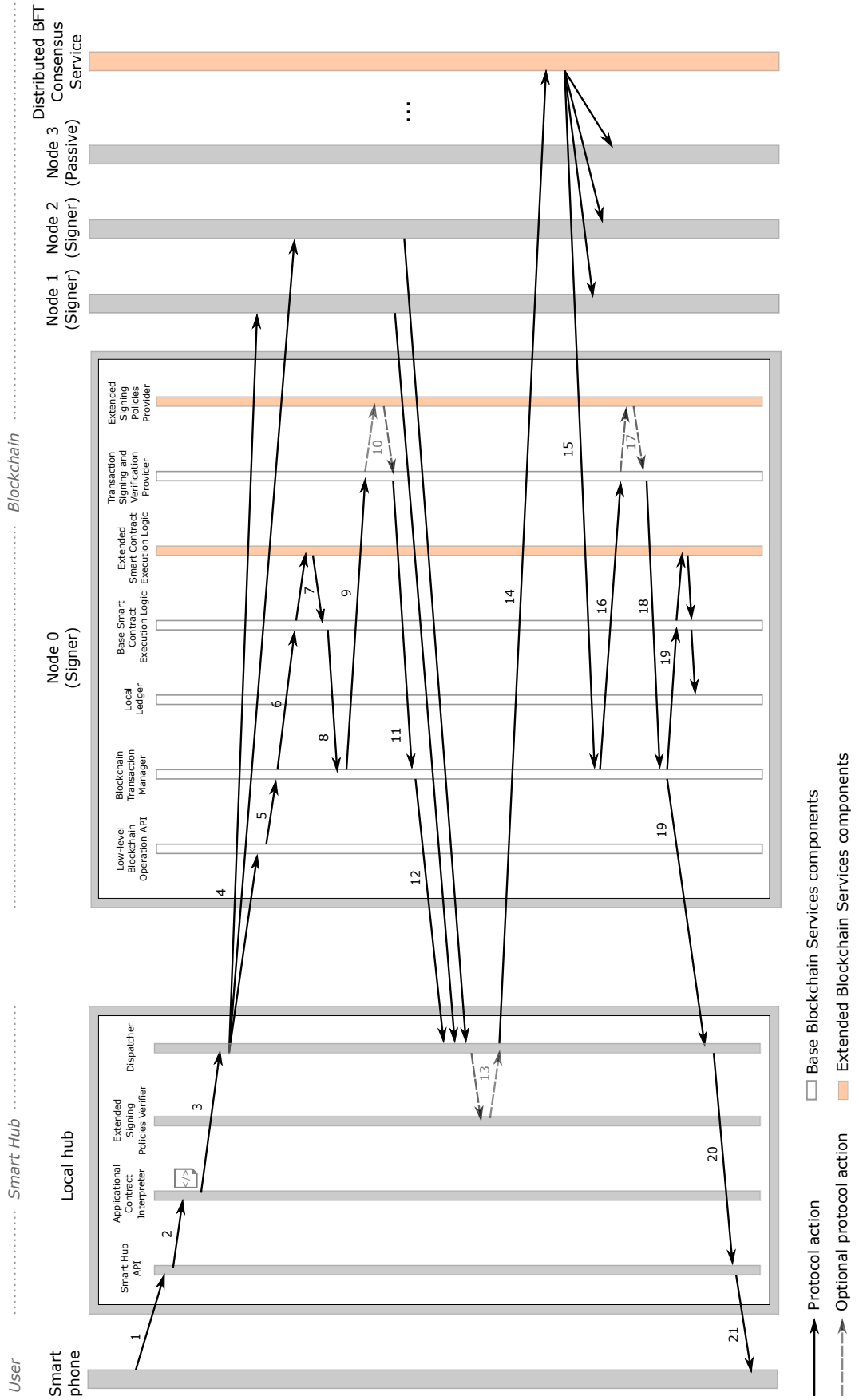

Figure 3.6: Flow diagram of a transaction proposal (write operation)

20-21. The result returned to the hub is forwarded to the user via API. In this context, an IoT device could read the result of the transaction executed by the user by either polling the hub or by receiving an event from the API.

Reading from the ledger. Figure 3.7 illustrates a flow diagram of the execution of the read protocol. The same conditions from the write operation apply here, i.e. assume the contract over which the read operation is to be executed was accepted beforehand by the user owning the IoT device in the illustration.

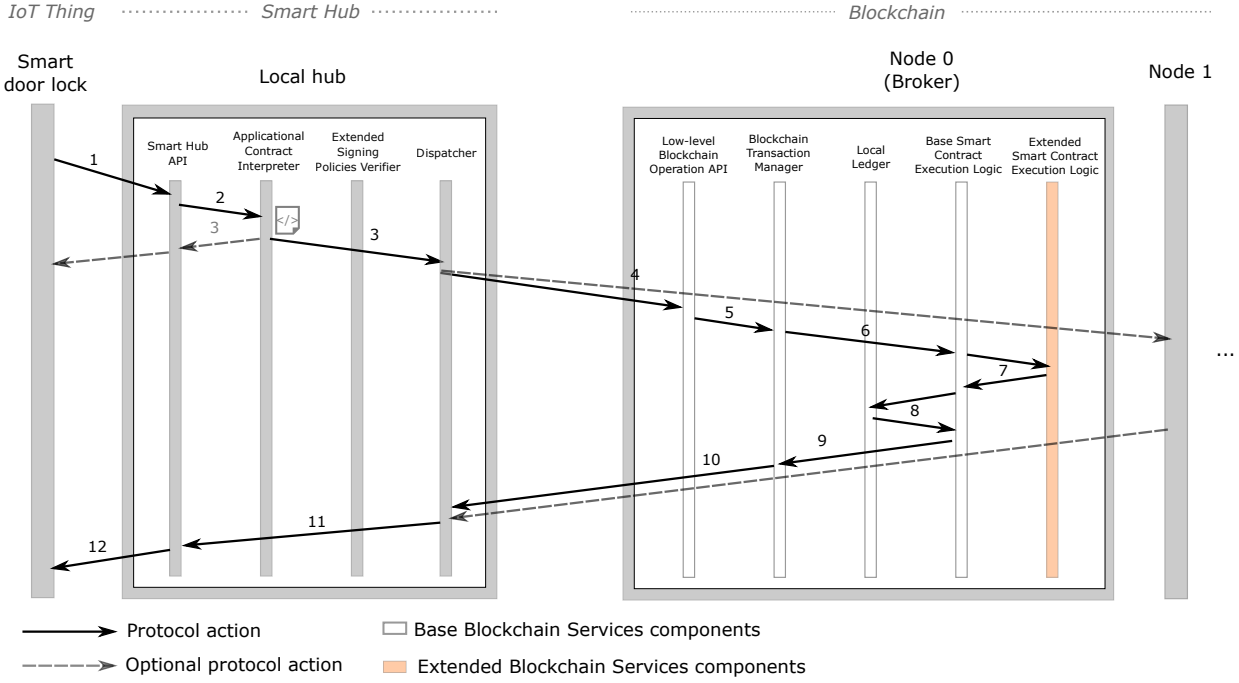

Figure 3.7: Flow diagram of a query over the ledger (read operation)

1. An IoT device queries the nearby hub it is linked to for some data, supplying a contract ID;
2. The smart hub, upon receiving the request, calls the applicational contract interpreter to verify any needed conditions to be ensured in order to successfully execute the read operation;
3. The contract interpreter reads the contract and checks whether it has any cached response for this request. If it does, it responds directly to the IoT device, else it passes any needed information about the contract to the dispatcher;
4. The dispatcher requests a read operation to the blockchain through the low-level API of a hub's broker node. In special cases, depending on the conditions of the contract, it may perform the same request to other nodes, whether they were a broker/bootstrap node for this device or not. For simplicity, let us assume the general scenario is just to query the broker;

5-7. The transaction manager receives the request and executes the read operation specified on the contract, which may invoke any needed extended execution logic to be concluded;

8-9. The contract reads data from the ledger and returns it to the transaction manager;

10-12. The read data is forwarded to the hub, and finally, returned to the IoT device.

3.7 Summary

In this chapter we presented our system model definition and architecture for a decentralized ledgering platform, regarded as a blockchain-enabled IoT platform, that is both reliable, scalable and independently auditable by any of the participating entities present at present at typical IoT environments: the users of a given service, the service provider, *things* and smart hubs. We present a model in which communication between these entities, in the form of transactions to a ledger, is mediated by a network built upon blockchain principles enriched with strong cryptographic controls for digital signatures that allow for robust decentralized trust and with a dynamically executing transaction flow regulated by extended smart contracts. This notion of extended smart contracts is a powerful one that allows regular blockchain contracts to be enhanced with the capability of changing how system-level properties of the blockchain are executed and allows them to be interpreted not only at blockchain-level, but also at application-level of software running on the participating entities.

We also define how smart hubs, either user-owned or service provider-owned, can be harnessed in conjunction with this ledgering platform to provide a scalable architecture capable of performing computations, aggregating things and users on the edge, interpreting and caching extended smart contracts and acting as representative proxies for resource-limited IoT devices. Thus, we believe this model follows the motto of this thesis – *to bring order into things* – in the sense that IoT environments can be structured into a more hierarchical topology, with a ledgering platform allowing for decentralized trust and regulating communication between all participating entities, rather than in peer-flat chaotic local networks where every device can talk to the next, while also mitigating the future scalability issues of today's cloud-first applications by employing aggregation on the edge with smart hubs and dispersing the blockchain network heterogeneously throughout cloud providers.

Considering the contents of this chapter, we then go through the discussion of the implementation of the discussed system model and software architecture, in addressing an prototype used for validation and experimental assessment.

Chapter 4

System Implementation

In this chapter we present the implementation details related to the prototyping effort of the system model discussed in Chapter 3. We start from the implementation overview, describing the technologies used, and then we present the software components and implementation options taken to build the software architecture of the prototype. An important consideration to bear in mind is that the prototype implements the base model of interaction of our system model, to conduct the experimental validation and assessment to be presented in the next chapter.

4.1 Prototype Overview and Technologies

Prototype Overview. As to provide a general overview of the implemented prototype, we can describe its composition by the following different software services and components:

- Blockchain-enabled services, leveraged and extended from the base Hyperledger Fabric (or HLF) components and modules, where the support for threshold and group multi-signatures, integrated with extended consensus plane services for Byzantine fault-tolerant properties are a relevant touchstone;
- The materialization of the smart hub component, focusing on its base assumptions for the intermediation of interactions from users and IoT devices (regarded as clients of the provided services in the blockchain-enabled architecture), and forwarding those operations as transactions with data-management functions enabled by the backed blockchain services;
- The implementation of a set of test and benchmarking tools, and demonstrative client implementations, for system validation, assessment and experimental observations.

Technologies. As stated before, for the base blockchain services layer of our prototype we chose to use the HLF blockchain platform, following the arguments discussed in Chapter 2. The HLF platform and its transaction flow mechanisms meet our system model, albeit some minor variations, offering an extensible leverageable base for our developments.

The prototype's blockchain services were built using version 1.1 of Hyperledger Fabric. The source-code of the platform itself is written in Golang 1.10, as are smart contracts definitions within the platform. As such, to implement our own custom smart contracts to be used within the platform and to modify the platform to our needs, we resorted to Golang as well. On some components of the platform, such as its consensus mechanism and digital signature components, the technology shifted to JAVA 9, with local communication between Golang processes and JAVA processes assured by UNIX domain sockets. Communications and messages exchanged with and within the HLF are done via gRPC[1] with Google's *protobuf* serialization[2]. Thus, when needed, we had to configure these and modify properties of the exchanged messages through parameterized Protocol Buffers.

The deployment of the blockchain platform was done by running a set of Docker containers, using Docker CE 18.03 and Docker Compose 2. Using Docker technology, we virtualize the blockchain network on a single machine, with each Docker container holding a blockchain peer process, allowing different containers to have different roles within the network. This platform can be deployed on a physically distributed environment, but for minimizing deployment overhead and quickly manage changes to the blockchain infrastructure, we preferred to follow a virtualized approach.

For the smart hub interface layer the technology used was JAVA 9, mostly due to the capability of writing a smart hub application that could run agnostically to the underlying hub's architecture and due to our familiarity with the language. To interact with the HLF we used the Hyperledger Fabric JAVA SDK 1.1[3], which we later modified for the needs of our system model. For local persistency on smart hubs, MongoDB 4.0[4] was used, as it is a lightweight key-value store. For exposing the hub in two distinct endpoints, one for constrained IoT devices and another for more capable devices leveraged by the end user, we resorted to a CoAP JAVA implementation provided by *mbed*[5], and Spark JAVA 2.7.2[6], a micro-framework for implementing RESTful applications harnessing a Jetty 9.4 HTTP web server[7], respectively. Communication between the hub and devices can be executed by exchanging messages in JSON format. Users can also interact with the hub in a web browser and obtain an HTML response. A fork of this layer intended at running a lightweight SSL library and DTLS [59] was also implemented using WolfSSL[8] (written in

[1] gRPC framework: https://grpc.io/
[2] Google Protocol Buffers (or *protobuf*): https://github.com/protocolbuffers/protobuf
[3] Hyperledger Fabric JAVA SDK: https://github.com/hyperledger/fabric-sdk-java
[4] MongoDB cross-platform document-oriented database: https://www.mongodb.com/mongodb-4.0
[5] CoAP JAVA implementation by *mbed*: https://github.com/ARMmbed/java-coap
[6] Spark JAVA framework: http://sparkjava.com/
[7] Jetty servlet engine: https://www.eclipse.org/jetty/
[8] WolfSSL embedded SSL library: https://github.com/wolfSSL/wolfssl

C) and the JAVA Native Interface (JNI)[9]. This fork, however, uses a custom binary format of structured byte arrays instead of JSON for exchanging messages.

Lastly, we also implemented an Android application in JAVA to provide a more realistic setting when interacting with the smart hub and to validate final end-user interactions via a smartphone. The targeted Android SDK of this application was version 28 and the minimum SDK was version 15, with the application able to run in Android 4.0.3 (Ice Cream Sandwich) as minimum requirement.

4.2 Prototype Architecture and Implementation

Mirroring our system model, our prototype implementation can be seen in layers: *i*) a Base Blockchain Services layer which we consider to be the HLF and the base services it provides; *ii*) an Extended Blockchain Services layer which extends the first layer. This layer is rather discussed as a conceptual layer, but we will refer to the two blockchain service layers as the whole extension of blokchain-enabled services, according to our development and as a new HLF-enhanced substrate; and *iii*) a Smart Hub Interface layer which has a direct mapping of the components in our system model. On top of that, we have got all of the entities required by our base model of interaction: users who will through some kind of medium (smartphones apps, browsers, or other client-side applications) interacting with smart hubs, IoT devices (or things), and service providers managing smart contracts on the blockchain and any nodes that may belong to them via an administration console or a back-office application. The architecture of the prototype is illustrated in Figure 4.1, which will accompany the remainder of this chapter.

In terms of implementation effort, the whole codebase of the prototype was implemented in around 5900 lines of code, as Table 4.1 shows. The Smart Hub Interface and the Android application, which were built from scratch and by integration of the aforementioned software libraries, resulted in around 2150 and 750 lines of code, respectively. The addition of components to the HLF and changes to its original codebase totaled around 3000 lines of code.

Component	**Estimated LoC**
Smart Hub Interface	2150
Overall implementation	2100
Changes to the HLF JAVA SDK	50
Blockchain Services	3000
Changes to the HLF platform	1200
Impl. of extended smart contracts	500
Rectifications to HLF BFT-SMaRt impl.	50
Impl. of the extended signing policies provider	1250
Android Application	750
Total	5900

Table 4.1: Prototype implementation extension metrics (LoC)

[9]JNI bridge for WolfSSL: https://github.com/wolfSSL/wolfssljni

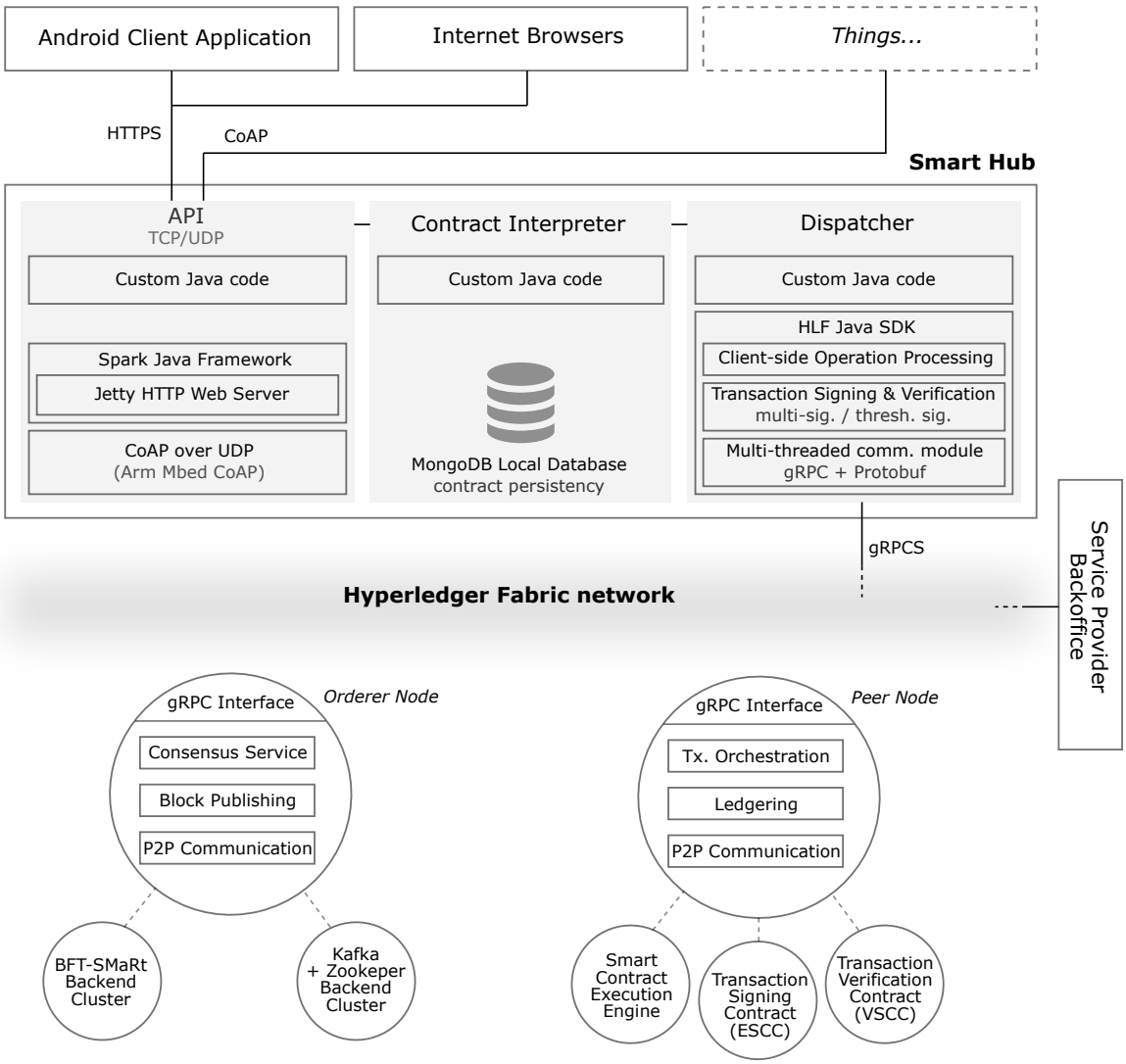

Figure 4.1: Prototype architecture

Regarding implementation complexity, developing and testing the prototype required the use of varied and distinct technologies, programming languages and language paradigms. Some of the technologies we used are in their infancy and yet to fully stabilize (e.g. the HLF SDK and APIs), and thus had higher than initially expected learning curves and unclear documentation. We harnessed multiple communication protocols (HTTP, gRPC and protobuf, CoAP, UNIX sockets) and were able to integrate such a wide array of technologies into a functional prototype composed by multiple artifacts. Moreover, we implemented the smart hub and extended blockchain service components as to be extensively configurable. The prototype was designed as a tiered architecture between clients, smart hubs and the blockchain, and fully implements the designed architecture from end-to-end. To do so, it required knowledge of different concepts: from blockchain architecture, server-side services, and network communication, to the intermediation of concurrent client communication through multi-threaded hub middleware, client-server

models with some level of data caching, and mobile development.

We emphasize two final issues as relevant implementation concerns in addressing the developed prototype. The first issue is the implementation strategy for reusing the base HLF system model components to leverage a pluggable model for a BFT consensus layer that includes the new types of group-oriented validation signatures; the second issue is the mapping strategy for smart contracts in the implemented prototype, using HLF Chaincode[10]. A Chaincode is triggered when a transaction is proposed and decides the state changes to be applied to the ledger. Thus, we must deal with initialization and management of ledger states by the received transactions, according to the new verification conditions, when group oriented multi-signatures or reliable threshold signatures are used, while maintaining the base assumptions of HLF, where Chaincodes can run in different containers from the peer and the state that is generated by a Chaincode is not accessible by another Chaincode, this way being able to maintain isolation guarantees.

4.2.1 Blockchain Services

To start explaining our implementation for the blockchain services layer we first have to give a brief description of the chosen platform: Hyperledger Fabric. We have already introduced the platform generically in Subsection 2.4.4, but for an easier comprehension we will provide a short summarized description of the platform relevant for explaining our implementation.

Base Blockchain Platform. The HLF is a platform which comes along with a few specific abstractions and concepts. Fabric is in essence, a permissioned consortium-based platform, whose consensus service relies on an off-chain model with fail-stop guarantees at most, i.e. it does not tolerate Byzantine behaviour. It is organization-oriented, which means that every node on the blockchain has a membership relationship with a single organization. Access control on the network is done by means of trust between organizations (through their respective certificates). Also, it uses the notion of channels, which can be viewed as partitions in a partial blockchain ledger replication scheme, i.e. transactions and contracts are only visible to an entity in the scope of a channel, if they have access rights to it. Within a single channel and a contract deployed on that channel, the HLF defines what is known as an endorsement policy, which is a set of rules composed by logical and-gates and or-gates where an administrator can specify the organizations that should sign transactions for that specific contract. Entities in HLF can be categorized into the following roles:

- *Peers*: Nodes responsible for maintaining a valid ledger of transactions and executing Chaincode logic. They are further divided into Endorser and Committer roles. As Endorsers, nodes endorse a given transaction by signing it after successfully executing it over a given chaincode without committing its results to the ledger

[10] HLF denotes its implementation of smart contracts as *Chaincode*, as referred in Subsection 2.4.4.

(imagine it as a simulation), while as Committers, nodes verify the integrity of the transaction and its appended signatures, and if successful, commit it to the ledger;

- *Orderers*: Nodes responsible for publishing blocks of transactions to Peers for them to be appended to the blockchain. To do so, these nodes resort to an Ordering Service to establish consensus on the order of transactions they receive from incoming messages.
- *Ordering Service nodes*: Nodes responsible for establishing consensus on the total history of transactions that occurred within the system. The only officially supported implementation for the HLF is an Apache Kafka and Zookeeper ensemble, a model which only provides crash fault-tolerant guarantees.
- *Clients*: Applications external to the blockchain which communicate with Peers and Orderers via the HLF SDK (e.g. our smart hub implementation).

Transaction flow for write operations, or invocations, within the HLF usually starts with a client application proposing a transaction to a set of Peers assuming the Endorser role. These nodes simulate the transaction proposal, endorse it if no invalid state occurs and return the signed proposal back to the client. The client can verify the signatures and the consistency of the responses and send the set of signed transactions to Orderer nodes. The Orderer delegates consensus to an Ordering Service and on its callback, compiles a result of ordered transactions into a single block. This block is then broadcasted across the network, where nodes, now acting as Commiters, verify the signatures of Endorsers, Orderers and the client, as well as the integrity of the payload. If successful, the block is then appended/committed to the ledger of each Peer. For read operations, or queries, what happens is that a client requests a query over the ledger to its known Peers, which then sign the response and return it to the client.

BFT Consensus Service. For this component, which is a part of our Extended Blockchain Services layer, we used the implementation by Sousa et al. [6, 7], which was, at the time of writing this document, a yet unofficial consensus service for the HLF. The service itself is implemented using BFT-SMaRt, a PBFT-like consensus mechanism, and is modeled so that HLF's orderer nodes possess a secondary process which allows them to proxy communication to a known set of BFT-SMaRt replicas. The communication between these two processes, the main orderer process and the proxy process, is assured in local memory by UNIX domain sockets.

The scope of the authors of the BFT-SMaRt implementation for the HLF was to create a consensus module that was functionally correct, robust and efficient. However, for the purpose of our implementation, we had to rectify and make some adaptations to the existing codebase as to allow multiple organization memberships to submit transactions to the consensus replicas as we need multiple organizations to have more than one signature per transaction. We also had to *dockerize* the two orderer processes within a single Docker image and create a set of bash scripts to allow configurability.

Extended Smart Contract. For the purpose of our prototype we implemented a contract which follows the Extended Smart Contract specification of our system model. We refer to this contract as XCC. The XCC was designed to be a generic smart contract with no specific business need in mind. As such, it only stores records in the form of a key-value store within the blockchain ledger. However, the more important part of the XCC is its ability to influence blockchain transaction flow and to be interpreted, albeit in a distinct representation and format, outside the blockchain.

Listing 4.1 is an excerpt of our XCC implementation that shows how properties of different sections of the contract – extended contract properties and application-specific properties – are structured and stored within the contract.

Listing 4.1: Excerpt of the XCC chaincode properties and functions

```
...
// Contract extended properties
type ExtendedContractProperties struct {
    ContractId          string      `json:"contract-id"`
    ContractVersion     int         `json:"contract-version"`
    AvailableFunctions  [][]string  `json:"available-functions"`
    InstalledOnNodes    []string    `json:"installed-on-nodes"`
    SignatureType       string      `json:"signature-type"` // multisig,threshsig
    SigningNodes        []Node      `json:"signing-nodes"`
    ConsensusType       string      `json:"consensus-type"` // bft, failstop
    ConsensusNodes      []Node      `json:"consensus-nodes"`
    ExpiresOn           string      `json:"expires-on"`
    ValidFrom           string      `json:"valid-from"`
    ProviderSignature   string      `json:"provider-signature"`
    DeployedOn          string      `json:"deployed-on"`
}

// Contract applicational properties
type ApplicationSpecificProperties struct {
    MaxRecords   int `json:"max-records"`
    TotalRecords int `json:"total-records"`
    // ...for example purposes only
}

// Records held on the ledger, i.e. actual data
type Record struct {
    Data string `json:"data"`
}

func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {
    args := APIstub.GetStringArgs()
    ...
    // Unmarshall args
    var extProps ExtendedContractProperties
    var appProps ApplicationSpecificProperties
    err := json.Unmarshal([]byte(args[0]), &extProps)
```

```
37      if err != nil {
38          return shim.Error(err.Error())
39      }
40      err = json.Unmarshal([]byte(args[1]), &appProps)
41      ...
42
43      // Save contract properties on the blockchain
44      extPropsCompositeKey, _ := APIstub.CreateCompositeKey("props",
45      []string{"EXTENDED_CONTRACT_PROPERTIES"})
46      extPropsAsBytes, _ := json.Marshal(extProps)
47      APIstub.PutState(extPropsCompositeKey, extPropsAsBytes)
48      appPropsCompositeKey, _ := APIstub.CreateCompositeKey("props",
49      []string{"APPLICATION_SPECIFIC_PROPERTIES"})
50      appPropsAsBytes, _ := json.Marshal(appProps)
51      APIstub.PutState(appPropsCompositeKey, appPropsAsBytes)
52
53      return shim.Success(nil)
54  }
55
56  func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response
57  {
58      // Retrieve the requested Smart Contract function and arguments
59      function, args := APIstub.GetFunctionAndParameters()
60      // Route to the appropriate handler function to interact with the ledger
61      // appropriately
62      if function == "getContractDefinition" {
63          return s.getContractDefinition(APIstub)
64      } else if function == "signContract" {
65          return s.signContract(APIstub, args)
66      } else if function == "getContractSignature" {
67          return s.getContractSignature(APIstub, args)
68      } else if function == "getEndorsementMethod" {
69          return s.getEndorsementMethod(APIstub)
70      } else {
71      ...
72  }
73  ...
```

System-level properties, the ones used by the blockchain platform, are not specified here as those are internal meta-data of the HLF. The properties stored directly on the contract are exportable in a JSON representation to the outside world (e.g. to the smart hub) and are populated when the contract is deployed to the blockchain by a service provider. In our prototype this is done via console commands by an HLF user with administration rights. The deployment of the contract consists of its installation on peers and then its instantiation, which results in a a call to the `Init` function seen below – the function that registers the contracts properties. The `Invoke` function, on the other hand, is the function executed when a query or an invoke operation is triggered over the contract. This function identifies the sub-routine it has to execute in order to fulfill the

call, reading or writing on the ledger.

Transaction Signing and Verification. We overrode two system chaincodes, the ESCC (Endorsement System Chaincode) and the VSCC (Verification System Chaincode). System chaincodes are a special type of contracts within the HLF that execute in their own dedicated environments (or containers) designed for executing system-level operations. Imagine them as a kernel functions of the blockchain. In this case, the ESCC is the chaincode responsible for endorsing transactions, i.e. signing them with a cryptographic algorithm, and the VSCC is responsible for verifying the integrity of the transaction and any signatures created by the ESCC.

Our changes to the ESCC and the VSCC were done in the scope of allowing the HLF to dynamically decide on what signature method to use to sign transactions over a given contract by querying the properties of that same contract. Logically, there is a property in the contract that defines the signature method to use. In our prototype, possible methods are: *i*) multi-signatures (the default HLF implementation); *ii*) and threshold signatures. The latter is provided by a module we developed called the Extended Signing Policies Provider (XSPP), which is an optional multi-threaded secondary process (in JAVA) running in the containers of Peer nodes, together with their main process (in Golang). We say optional, because this approach allows a modular implementation; nodes that do not need to endorse transactions will probably not need to boot the XSPP. Communication between these two processes is done via UNIX domain sockets, similarly to the aforementioned BFT-SMaRt service, in TCP streams. Each new connection to the XSPP will be attended by a thread from a fixed but configurable thread-pool in the XSPP. The reason for the existence of this module as a separated component from the HLF main process is, first, because it allows for further extensibility, i.e. in the future, new policies and signature schemes other than threshold signatures can be added to this module without having to disrupt the codebase of the HLF Peer node, and second, because Golang has a lack of cryptographic primitives support for threshold signatures (possibly due to the recency of the language), namely for Shoup's threshold signature scheme [67], which is the one we intended to implement.

The implemented signing and verification processes of the prototype can visualized in Figure 4.2. The original signing processes consisted of the signing node simply executing a given operation over a specific chaincode and then signing it with its organization signature. If the client sent transactions to multiple Endorsers, the result would be a multi-signature scheme. Regarding verification, the process was similar, verification of signatures was done using the multi-signature scheme by default before an integrity validation of the transaction against the contract. In our implementation, a different protocol occurs.

For signing transactions (top diagram), the implemented process is as follows:

1-2. After an Endorser receives a transaction proposal from a client, it verifies the transaction against the contract, which runs in an isolated chaincode environment (or

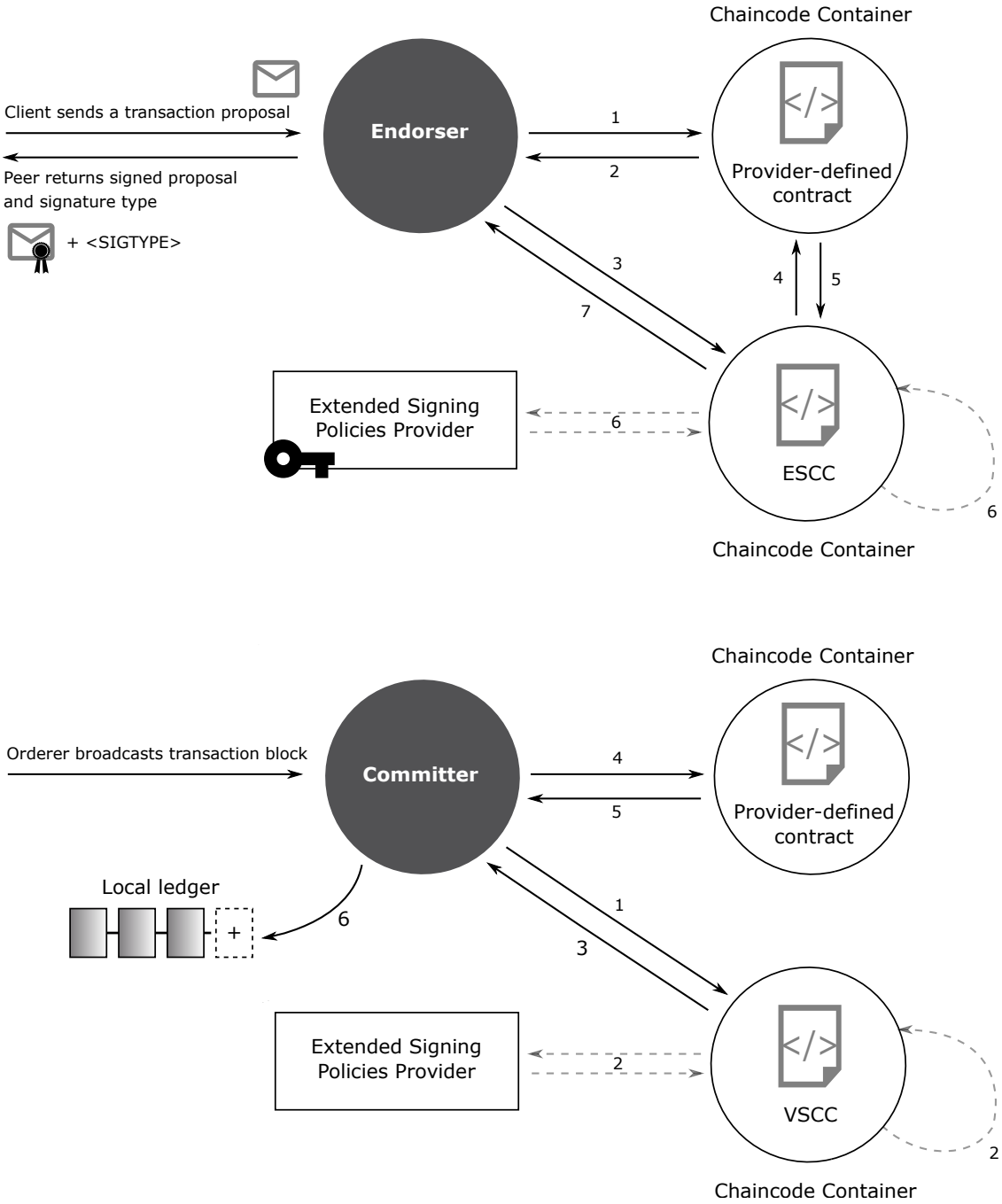

Figure 4.2: Signing (top) and verification (bottom) of transactions

container);

3. After a successful contract validation, the node requests the ESCC to sign the transaction;

4-5. The ESCC queries the properties of the contract in a chaincode-to-chaincode invocation;

Note: Querying the signature method and properties from the contract is slow, requiring a heavy number of internal validations and calls, and is done only for the first transaction over the contract. The ESCC keeps record of the signature method for posterior transactions;

6. If multi-signature is the method specified on the contract, the ESCC uses the default signing mechanism. If not, it requests the XSPP to sign the transaction instead.

7. The signed transaction is returned to the node, which registers the signing method on the transaction payload, and is sent back to the client.

For verifying transactions and consequently committing data to the ledger (bottom diagram), the implemented process is as follows:

1. After a Committer receives a transactions block from an Orderer, it requests the VSCC to verify the appended signatures, which it does by reading the signature method property on the payload of the transactions to understand which method of signature verification it should use;

1. If the VSCC identifies that the method used to sign the transactions was a multi-signatures scheme, it uses the default signature verification scheme. Otherwise, it requests the XSPP to verify the signatures instead;

3-4. After successful signature validation, the transactions are verified against a given contract;

5-6. Upon a successful contract validation, the Commiter records the transaction block and its signatures to its local ledger;

XSPP Implementation Details. The XSPP implemented on our prototype, albeit its extensibility, harnesses only a threshold signature library at its latest version. The library integrated into the solution was the one in [73]. Outputs and inputs of the library were originally defined in a numeric format, i.e. group keys, private key shares and signature shares were all defined by the mathematical notations of RSA cryptographic materials (e.g. 64-bit integers for exponents). We defined a custom serialization format for these objects where we build a byte array structured into different fields, each one for each parameter of the object, and then converting the whole array into a base-64 representation which could

be used for communication purposes with peer nodes. For the purpose of our prototype, we inject key shares and respective group key into peer nodes as environment variables.

The library also had a limitation in which it would require exactly K signature shares to be fed into its verification function or it would always result in a failure, K being the minimum quorum size of a threshold signature scheme. We believe that for any $L \geq K$, the verification function should always be able to validate the transaction assured the threshold of valid signatures is met, so we implemented an iterative permutation algorithm within the verification function. Essentially, given an $L \geq K$ to the verification function, the algorithm generates a permutation of K signature shares and then proceeds to verify the function. If it fails, it repeats the process for a new permutation. If all permutation-verification pairs fail, then the threshold signature verification fails permanently. Theoretically, in the best-case and most likely scenario, the complexity of this approach is just of generating a single permutation and verifying it immediately. In the case that there are enough Byzantine parties to corrupt the signature, the verification process will run through all possible permutations until it actually concludes the signature was corrupt, resulting in a possible performance degradation in transaction verification. Thus, going back to our system model, in Subsection 3.4.3, we can classify our threshold signature verification scheme as optimistic and synchronous, since we verify immediately the first given k signatures and expect in the best case scenario that there is no corrupted signature present. It is synchronous as the algorithm itself makes the requester block while waiting for verification to complete.

4.2.2 Smart Hub

As can be seen in Figure 4.1, our smart hub implementation consists of three major components: the API, the Contract Interpreter and the Dispatcher. In effect, these components were originally outlined and specified for the Smart Hub Interface in our system model, but all other components of that same layer are present here, even though they might be represented as a minor part of the smart hub architecture. These three components can be seen as a sequence of states for requests sent to the hub: requests enter through the API, where they are routed, verified in terms of correctness of the request (if the action the request is trying to trigger exists, if all mandatory parameters are supplied, if a client certificate is supplied when TLS mutual authentication is required, and any other needed validations), unmarshalled (if any payload exists), and passed on to the Contract Interpreter. The Contract Interpreter verifies the contents of the request against a given contract specified by the request and then passes the results of interpreting the contract's properties and the contents of the request to the Dispatcher, which orchestrates all communication with the blockchain services according to the supplied information.

Furthermore, given that current devices close to the notion of a smart hub [2, 3, 61] are very heterogeneous at both hardware and operating system levels and that the rising trend of the IoT is only expected to increase, we intended with this implementation

to provide a smart hub interface which would be able to be executed on most devices independently of their hardware or OS specifications. However, as this heterogeneity may lead to different resource capabilities, we intended to develop the hub with as lightweight components and libraries as reasonably possible. Thus, our choices for the technologies of the smart hub implementation were heavily influenced by these premises.

URL: `https://<address>:<port>/api/<channel>/contract/<contract-id>` *Method:* `GET` *Response format:* `application/json` or `text/html`, depending on an Accept header.
Returns a representation of the contract with identifier equal to `<contract-id>`, if it exists within the blockchain and within channel `<channel>`. It also returns a SHA256 hash of the contract for users to sign and accept the terms of the contract.

URL: `https://<address>:<port>/api/<channel>/contract/<contract-id>/sign` *Method:* `POST` *Response format:* `application/json` or `text/html`, depending on an Accept header. *Requires:* A payload with a client signature of the contract.
Allows the user to accept the contract with identifier equal to `<contract-id>`, if it exists within the blockchain and within channel `<channel>`. Returns a confirmation or an error back to the caller.

URL: `https://<address>:<port>/api/<channel>/contract/<contract-id>/query` `coap://<address>:<port>/contract` *Method:* `POST`* *Response format:* For HTTP, `application/json` or `text/html`, depending on an Accept header. For CoAP, only `application/json` is supported. *Requires:* A payload with the function name to query and optional arguments. For CoAP, `<channel>` and `<contract-id>` have to be prepended to the payload.
Allows a query to the service to be executed over the contract with identifier equal to `<contract-id>`, if it exists within the blockchain and within channel `<channel>`, and if the supplied function name exists. Returns the query results or an error back to the caller.

URL: `https://<address>:<port>/api/<channel>/contract/<contract-id>/invoke` `coap://<address>:<port>/contract` *Method:* `POST`* *Response format:* For HTTP, `application/json` or `text/html`, depending on an Accept header. For CoAP, only `application/json` is supported. *Requires:* A payload with the function name to invoke and optional arguments. For CoAP, `<channel>` and `<contract-id>` have to be prepended to the payload.
Allows an invocation to the service to be executed over the contract with identifier equal to `<contract-id>`, if it exists within the blockchain and within channel `<channel>`, and if the supplied function name exists. This invoke operation writes over the ledger by sending a transaction to the blockchain. Returns a confirmation or an error back to the caller.

* The operation can be called with `GET` on browsers to get an HTML form for executing the `POST` method.

Table 4.2: Smart Hub API operations

API. Our implementation of the smart hub API exports a set of HTTP/CoAP operations which all map to the stub defined in our system model in Section 3.4. We segment traffic between these two protocols and expect users, through their smartphones or other medium, to contact the hub via HTTP, and IoT resource-constricted devices to contact the hub via CoAP. As such, operations related to obtaining and accepting a smart contract are only supported via HTTP since these are only relevant for users of the service. Both HTTP and CoAP servers are multi-threaded while attending requests. All supported operations are described in detail in Table 4.2, where the fields `<address>`, `<port>`, `<channel>` and `<chaincode>` represent the Internet address and port of the smart hub, and the HLF communication channel where a given smart contract is deployed, respectively.

Contract Interpreter. A local MongoDB instance runs within the smart hub for the purpose of providing a local smart contract persistency mechanism. This instance keeps contracts cached on the hub and retrieves them from persistency by their identifier. When an operation over a specific contract is requested to the Interpreter, there is a first attempt to locate in cache. If unsuccessful, the Interpreter retrieves the contract from the blockchain and stores it in the local store for future operations, only then proceeding to extract and validate the contract's properties. No operation is executed without verifying the respective contract's properties at least once.

The Contract Interpreter validates a set of properties on the contract in a fashion that resembles the validation of X509 certificates. It first identifies if the contract has a given standardized structure (specified in Subsection 3.4.1), and then proceeds to verify fields such as the expiry date on the contract, the beginning date, if the contract has clearly defined the signature method and consensus service endpoints to be used, and so on. After successful validation, properties such as the nodes that will sign the transaction, which signature method to use to verify signatures obtained from blockchain nodes are passed on to the Dispatcher component. This validation is only executed for each contract per run of the smart hub, for performance reasons.

Dispatcher. The implemented Dispatcher component as a whole has a few functions: *i*) it contacts a set of boostrap nodes upon booting up the smart hub in order to get any requested contracts from a user from them; *ii*) it initializes and maintains a set of Fabric channels which the hub is authorized to communicate with; *iii*) and it triggers query and invoke requests to peer nodes; the operations of the API, while having additional complexity added to them, all delegate to these two last functions. This last component of the smart hub depends heavily on the HLF SDK library, which is responsible for sending and receiving requests from the HLF network using the gRPCS protocol with protobuf serialization of messages. The SDK is essentially a multi-threaded client capable of doing some client-side operation processing, and together with the gRPC interface on each blockchain node, it implements the brokering mechanism.

As the HLF verifies transaction signatures in two points, one in the blockchain network before committing transactions to the ledger, and another one on client applications

using the HLF SDK when transaction responses are received from Endorsers, the SDK was modified by us to support dynamic switching of signature verification schemes according to the properties of a contract. We implemented a mechanism that verifies a property on transaction proposal responses set by Endorsers stating the signature method used to sign that response, so that the SDK can decide which signature verification scheme to use. For the multi-signature signature scheme, the SDK resorts to the default implementation. For threshold signatures it resorts to the library in [73] that we integrated for that purpose. After successful verification and checking peer responses for coherence, the SDK produces an envelope where it also registers the signature method used and sends it to orderer nodes.

4.2.3 Other Implementation Aspects

Android Client Application. As mentioned before, we implemented a small Android application for an increased degree of realism while interacting with a smart hub, either emulated on a computer or running in a dedicated physical device. The application implementation is quite simple and consists of three screens (or activities in Android slang): *i*) a configuration activity for users to input needed parameters to communicate with the hub (hub address and port, channel, contract name); *ii*) an activity for visualizing contract specifications and accepting them; *iii*) and an activity for executing query and invoke operations over the hub. All communications with the smart hub, protected with TLS 1.3 and mutual authentication, result in the application calling the operations in Table 4.2. For accepting a contract, something which just requires a press of a button from a user, the Android application employs an automated signing mechanism, using its private key to sign the hash of the contract, returning it to the hub.

DTLS Smart Hub API. We implemented an alternative API version, which uses a lightweight WolfSSL library for DTLS [59] communication with external entities, both users and IoT devices. It exports the same functions as the original implementation, albeit in a different format. A structured byte array is used to send messages back and forth between users and things in communication with the hub, where the first section of the byte array is an integer representing the operation being executed and the second section is a set of arguments needed for correctly executing the function. This version, however, was not explored as much as the dual protocol HTTP/CoAP version.

4.3 Summary

In this chapter we described the implementation effort related to building a prototype based on the system model of the previous chapter. We highlighted and went into detail where needed of the main features of the prototype, namely: the usage of Hyperledger Fabric as the base blockchain platform, the integration of the BFT-SMaRt consensus service into the platform, our implementation of extended smart contracts which can be

dynamically interpreted at different levels of our model, how that same interpretation influences transaction and verification processes within our system and how do we integrate threshold signatures into these processes, our implementation and architecture for smart hubs and all respective components and other relevant implementation aspects, such as an Android application used for a more realistic end-to-end testing in a client–smart-hub–blockchain topology.

The prototype was built upon the base model of interaction where smart hubs intermediate communication of users and things with the blockchain services, it is extensively configurable, both the smart hub and the blockchain platform, and was written in around 5900 lines of code. Its source-code shall be available in the open-source community[11].

A final consideration to bear in mind is that our current prototype implements the relevant base assumptions in the model of interaction referred in Chapter 3, offering a *pluggable* base for other future developments. Given time restrictions, implementing a more extended model of interaction, namely supporting other IoT edge-based ecosystems and their protocols, enhanced smart hub functions, and integration of other cryptographic methods for threshold-based signatures, with expressive definitions for the execution of smart contracts, would prove itself complex and require a higher amount of work. This would be difficult to fit within the scope of this thesis. However, we believe that the modular approach of the implemented prototype offers the possibility to capitalize this effort, as interesting future work research directions.

[11] Prototype source-code at: https://github.com/fmiguelgodinho/dsliot-prototype-difctunl

Chapter 5

Experimental Evaluation and Analysis

In this chapter we describe our experimental assessment effort over the developed prototype described in Chapter 4. First we describe our test-bench environment conditions, and then we proceed to present our obtained results from evaluating each prototype component. We analyze results in detail, relate them to our expectations according to our system model and related work, and identify open issues where applicable.

In summary, our evaluation criteria, i.e. the questions we answer with the analysis of the experimental results in this chapter are the following:

- Can we build a blockchain-supported system with Byzantine fault-tolerance guarantees and decentralized trustability assumptions with a viable throughput[1] and with a security level on pair with modern security standards?
- Can we further decentralize and increase the robustness of blockchain architectures by modifying transaction signing and verification processes to be group-oriented and fault-tolerant while maintaining an acceptable throughput?
- Can different signature schemes for blockchain transactions work better under different parametrization conditions?
- Can we provide a scalable edge-based smart hub architecture for the IoT capable of reasonable throughput under increasing stress conditions?
- Can we provide a system model where protocol and communication weight on IoT things, sensors and actuators is minimal?

[1] By *viable throughput* in BFT guarantees with decentralized trustability assumptions, what we mean is that we wish to assess how adding BFT consensus and threshold signatures to Hyperledger Fabric can affect the practicability of the solution and of the theoretical throughput referred in Section 2.4.

5.1 Test-bench Environment

The test-bench environment for evaluating our implementation, and consequently our system model, was set up in the below topology. In terms of technical specifications for each machine, their characteristics are summarized in Table 5.1.

- A dedicated cloud server for hosting the blockchain services environment, in which blockchain nodes ran in virtualized Docker networks;
- A single-board Raspberry Pi computer for running the smart hub API and closely emulating a real-world scenario of a physical smart hub deployment in which it would be expected to have as less complex hardware as possible;
- A laptop used for emulating the smart hub API as an alternative to the Raspberry Pi above, in order to assess the smart hub API in optimal resource conditions;
- An HTTP and CoAP client built by us and executing from the laptop above for benchmarking an end-to-end scenario between clients and the blockchain intermediated by smart hubs. In effect, benchmarks from this tool were measured directly against a single smart hub, either deployed in the same machine (emulated deployment) as the benchmarking client or on a Raspberry Pi on the LAN;
- A gRPC client – Hyperledger Caliper[2] – for benchmarking the blockchain services network with asynchronous workloads, running on the same laptop as the above client. This client used a variation of the HLF SDK 1.1.0 to communicate with the HLF network, a version implemented in *node.js*;

	Dedicated cloud server	Benchmark client / Smart hub	Smart hub
CPU	Intel Xeon D1520 @ 2.2-2.7GHz	Intel Core i5-5200U @ 2.2GHz	Cortex-A53 @ 1.4GHz
RAM	128Gb DDR4 2133MHz	16Gb DDR3 1600Mhz	1Gb LPDDR2 SDRAM
OS	Debian Stretch 9.4	Windows 10 Pro	Raspbian Stretch lite 4.14
Model	OVH HOST-128L	HP EliteBook 840G2	Raspberry Pi 3B+

Table 5.1: Technical specifications of the test-bench environment

We measured memory consumption and CPU usage for the smart hub API by performing real-time monitoring of the application using JConsole, and produced runtime heap dumps that were then analyzed using Eclipse MAT[3].

The hardware components in this topology *at the edge*, i.e. our testing local laptop and the Raspberry Pi, were linked by a single switch in a dedicated network environment as to reduce interference between network traffic of communications related to our system with traffic produced by other applications, which could negatively impact the benchmarks. The switch was connected to the Internet by acting as a DHCP client for

[2]Hyperledger Caliper: https://www.hyperledger.org/projects/caliper
[3]Eclipse MAT: https://www.eclipse.org/mat/

an ISP-provided switch via Ethernet with a measured connection of 100 Mbps. WLAN was measured at around 10 Mbps. HTTP and CoAP requests between the smart hub benchmarking client and the Raspberry Pi were executed over WLAN, while communication between the Pi and the dedicated cloud server were executed over Ethernet links, as the Pi was directly connected via Ethernet to the dedicated switch. The HTTP connection between the benchmarking client and the smart hub API was protected by TLS with mutual authentication. Benchmarking the blockchain platform with Hyperledger Caliper was performed via Ethernet. At a virtual network level of the dedicated cloud server, the Docker network supporting the blockchain platform was segmented into two distinct virtual subnets: one for consensus nodes (BFT-SMaRt replicas, Apache Kafka and Zookeeper instances), and another one for regular blockchain nodes (peers and orderers).

In terms of network latency between the test hardware, the measured ICMP ping RTT latency from the benchmarking client running on our test laptop and the Raspberry Pi to the dedicated cloud server was of $47ms$, and the latency of $< 1ms$ was measured between the client and the Raspberry Pi.

5.2 Benchmarks and Analysis

For the following benchmarks, we start with the observation of the baseline capabilities of our testing cloud-based infrastructure for hosting the Hyperledger Fabric. We then evaluate the base platform and our incrementation of functionality over it, i.e. our prototype blockchain platform. Afterwards, we proceed to the evaluation of our developed smart hub API prototype.

In all benchmarks ahead, we either directly evaluate the developed prototype blockchain platform or depend on it as a supporting asset of smart hubs. Thus, it is worth to note that for our blockchain network, we assume a total of l nodes, out of which n are endorser nodes. Out of these n endorsers, there may be a number of f faulty nodes within the network. Excluded from l are nodes related to consensus: HLF orderer nodes, BFT-SMaRt replicas, and Apache Kafka and Zookeper instances. These were generally locked at 4 orderer nodes, 4 BFT-SMaRt replicas, 4 Kafka instances and 3 Zookeeper nodes. The presence of BFT-SMaRt replicas and Kafka and Zookeeper nodes was mutually exclusive to whether BFT was required for consensus. Except for benchmarks where we explicitly varied the size of ledger block sizes, we set block size to a maximum of 10 transactions. Block interval is set at $2s$ to prevent flooding peers with transaction blocks. All our tests are executed over a single HLF channel. For experimental assessments where threshold signatures are leveraged, we assume $n = 3f+1$, f being the maximum tolerable faults, and set $k = f+1$, k being the minimum viable number for threshold signature reconstruction quorum size. We request an odd number of signatures in order to obtain a majority in signature verification quorums.

Regarding signature algorithm parametrization, we perform tests on RSA threshold signature schemes with varying modulus values $N = \{1024, 2048, 3072\}$ and on ECDSA

multi-signature schemes with a modulus size of $N = 256$, which are harnessed by the original HLF implementation. Our rationale behind the variation of RSA modulus values follows on NIST guidelines[4]. In summary, key lengths of 2048 bits are reasonable for today's standards up until 2030; from that moment on 3072 modulus sizes should be used for RSA schemes. Keys with a length of 1024 bits are considered to be insecure and should only be used by legacy systems where it is infeasible to upgrade. In terms of security level, we establish equivalence between the RSA and ECDSA signature schemes to be roughly at the 2048 to 3072 bits in RSA versus 256 bits in ECDSA.

It is also worth to note that tests against the HLF blockchain platform, either the original or our prototype, benchmark the full lifecycle of a transaction: i) from when they are proposed by a client to a set of endorser nodes; ii) to the moment they are verified against a smart contract, signed (endorsed) and returned to the client; iii) to the client receiving all needed endorsements and sending the transaction to the HLF ordering service; iv) to the ordering service establishing consensus via some off-chain consensus cluster and publishing a transaction block to the network; and finally v) to the moment peer nodes verify, commit the transaction to the ledger and confirm it to the client. For tests against smart hubs, this cost is also included in our benchmarks.

5.2.1 Baseline Observation

Before we assess Fabric in a distributed setting, let us first infer the practical maximum throughput of our testing machine. Table 5.2 summarizes the results of two micro-benchmarks we executed initially for this purpose. The first baseline measure is a minimal run of the HLF as a centralized system where there is a single peer for endorsing transactions and a single orderer for publishing transaction blocks. This test was executed with a payload of 100 asynchronous invocations of write operations over a single chaincode. The second baseline is a set of measures obtained from benchmarking the HLF ordering service only, without any peer nodes involved. This benchmark solely assesses the ability of a BFT HLF ordering service to close transactions (with a size of 1Kb and 4Kb), i.e. to receive them, establish consensus, put them into a block and start propagating them across the network. The benchmarking client produced a workload of 10000 signed transactions with a random payload for each run. It is important to notice that this measure does not benchmark the full lifecycle of transactions within the HLF and is the only exception to this rule in our benchmarks.

This assessment is important so that we can understand the difference in throughput metrics for the HLF, which for its ordering service can reach orders of magnitude of hundreds or thousands of transactions and for the full lifecycle of transactions may be far less, especially taking into account we are running Fabric in a single machine and not in a physically distributed environment. It is also an important observation of the

[4]See NIST's guidelines for cryptographic key lengths here: https://www.keylength.com/en/4/

Baseline measure	Replicas (r)	Throughput (tps)
Original HLF with 1 orderer and 1 peer node (no off-chain consensus)	-	15
BFT HLF ordering service [6] with 1 orderer node	4	1082 (1Kb) - 871 (4Kb)
BFT HLF ordering service with 1 orderer node	7	623 (1Kb) - 483 (4Kb)
BFT HLF ordering service with 1 orderer node	10	332 (1Kb) - 268 (4Kb)

Table 5.2: Baseline of the Hyperledger Fabric and impact of the BFT ordering service

decay in performance of the BFT ordering service, which uses the BFT-SMaRt implementation, with an increasing number of consensus replicas r and when transaction payload increases. We benchmarked the BFT version of the ordering service and not the original ordering service because a benchmarking client we could use for this purpose was readily available[5]. For the official ordering service, as we will see in the tests ahead, we can extrapolate throughput to be greater.

5.2.2 Base Platform Throughput

In this first benchmark, our intention was measure the average latency and throughput of the original HLF 1.1 blockchain platform with a varying network size and a varying number of endorser nodes. This test allowed us to obtain a criteria comparison of expected performance that can be used for all posterior benchmarks that involved our prototype.

In terms of test conditions, we initialized a single Hyperledger Caliper client and submitted Fabric to a workload of 100 asynchronous transactions by invoking write operations over the same chaincode while varying the total number of peer nodes present in the network to $l = 10, 20, 30, 40, 50$. The number of endorsers was also increased accordingly to $n = 5, 7, 11, 13, 15$. The write operation being invoked was equivalent to a simple insertion in a key-value store and its payload was randomly generated.

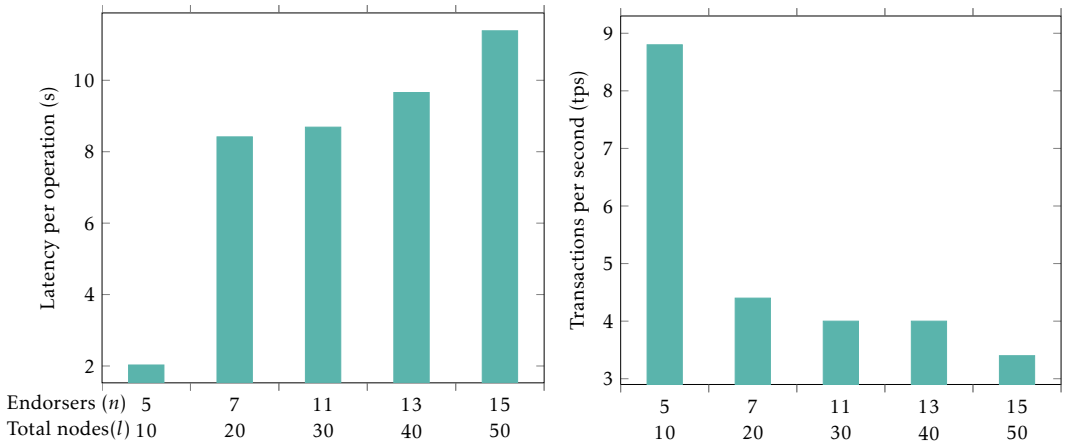

Figure 5.1: Base blockchain platform latency (left) and throughput (right) assessment with a varying number of endorsers and total nodes.

[5]BFT-SMaRt ordering service and client: https://github.com/bft-smart/fabric-orderingservice

Our obtained results are illustrated in Figure 5.1. From these results we can see a very high increase in latency per operation and a steep fall of throughput from when the network is composed of just 10 nodes with 5 endorsers to when the network size is doubled and 2 more endorser nodes are leveraged. Curiously, this performance hit seems to have a increasingly lesser impact when we keep expanding the network and the number of endorsers.

Looking back to our baseline assessment in the previous section, we had a measure of 15 tps with a single peer and single orderer. The two initial measurements of this benchmark, output around 8.8 and 4.4 tps. For these three initial conditions, performance degradation seems to be exponential. Given this loss of throughput was essentially caused by increasing network topology, the number of endorsers, and that the ordering service of the HLF may reach orders of magnitude of hundreds or thousands of transactions per second, this leads to our belief that the root cause for this behaviour may be the HLF's gossip algorithm, leveraged by endorsers to converge on ledger state. We suspect the temporal complexity for exchanging messages with this protocol may reach $O(n^2)$. This hypothesis is corroborated by the author in [63], where similar results are obtained.

Overall, the results we obtained for $l \geq 30$ and $n \geq 11$ were unexpected as we were expecting a higher performance hit for networks of these sizes. We identify this assessment to be an open issue, as we were not able to evaluate the root cause behind this behaviour. We suspect on the gossip between endorser nodes to be the issue behind the initial decrease of performance, which should continue to exponentially decrease throughput for the aforementioned network sizes. A dedicated evaluation to address this issue is suggested to be performed in future work.

We also find that our HLF throughput results may seem low at first (almost reaching 2 tps in some cases), especially when compared with benchmarks such as the ones in [72]. However, we remind the reader that our setup is a virtualized Docker network on a single machine, while, for instance, in [72], the authors are evaluating the HLF in a physically distributed setting in a high-end datacenter with 1 orderer node, an Apache Kafka and Zookeeper ensemble of unknown size, and 8 peer nodes, of which only 2 are endorsers, each instance running isolated in its own physical host and without sharing server resources. Given a powerful enough production infrastructure, the performance of HLF can theoretically reach over 1000 tps as mentioned in the related work.

5.2.3 Prototype Throughput and Comparison of Signature Schemes

For this benchmark, our objective was to evaluate the throughput of our prototype blockchain platform with Byzantine fault-tolerance consensus guarantees and distinct transaction signature schemes: multi-signatures and threshold signatures. Moreover, as aforementioned, we conducted our tests with varying RSA modulus values for threshold signatures in order to assess the impact related to increasing key sizes on the overall throughput of the platform.

Regarding test conditions, similarly to previous benchmark, this test was performed using a single Hyperledger Caliper client issuing 100 asynchronous transactions to blockchain nodes invoking write operations over an extended smart contract which defined the signature process to use. We launched a pre-emptive set of requests to all nodes before starting the evaluation. This ensures three things: *i*) that chaincode containers start running before the test (the HLF lazy loads chaincode containers; thus, there is a performance impact on our tests if we let containers boot mid-experimentation); *ii*) that blockchain nodes are ready to execute operations and that no invalid state exists; *iii*) to allow the ESCC to identify the signature method to be used for the contract in testing, minimizing time.

Before we analyze the impact of different transaction signature schemes, let us first highlight the impact of replacing the HLF original ordering service, which is tolerant to faults in a fail-stop model, with the prototype BFT ordering service from Sousa et al. [6] in the same network topology conditions and with the same n endorsers for signing transactions: $l = 20, n = 7$. Table 5.3 summarizes this aspect.

Ordering service	Consensus model	Throughput (tps)
Apache Kafka and Zookeeper (4 + 3 instances)	Fail-stop	4.4
BFT-SMaRt (4 replicas)	BFT	3

Table 5.3: Throughput comparison between the default and BFT ordering service

The performance decrease in throughput seen here from using a consensus mechanism that provides Byzantine fault-tolerance guarantees versus one that only supports crash faults is somewhat expected. The underlying BFT-SMaRt ordering service implementation, which runs a BFT SMR consensus algorithm, has to wait for a quorum of $3f + 1$ responses for each consensus round. This may result in an increment in overhead related to providing resilience in the possibility of Byzantine faults, in comparison with Zookeeper's algorithm, Zab, which establishes quorums at $2f + 1$ correct responses. With this information in mind let us proceed with the introduction of signature schemes.

Illustrated in Figure 5.2 are the results for this benchmark. The first information we can withdraw from these results is that the original HLF ECDSA multi-signatures are relatively inexpensive in throughput and latency when compared to RSA threshold signature schemes. When subjected to scalability conditions in the number of endorsers, the impact on threshold signatures and the throughput difference between both schemes becomes increasingly accentuated. When compared with RSA threshold signatures of a similar security level – 2048 to 3072 bits – ECDSA multi-signatures of 256 bits outperform such schemes by a considerable magnitude. The increase in latency between different RSA modulus values N can justified by the exponentiation operations to N in signature verification which weighs in on the protocol. However, bear in mind that the performance hit of threshold signatures seen here is a trade-off for decentralization, robustness and smaller transaction payloads, as we will see in further benchmarks. Given that our implementation is based on the optimistic version of the threshold signature

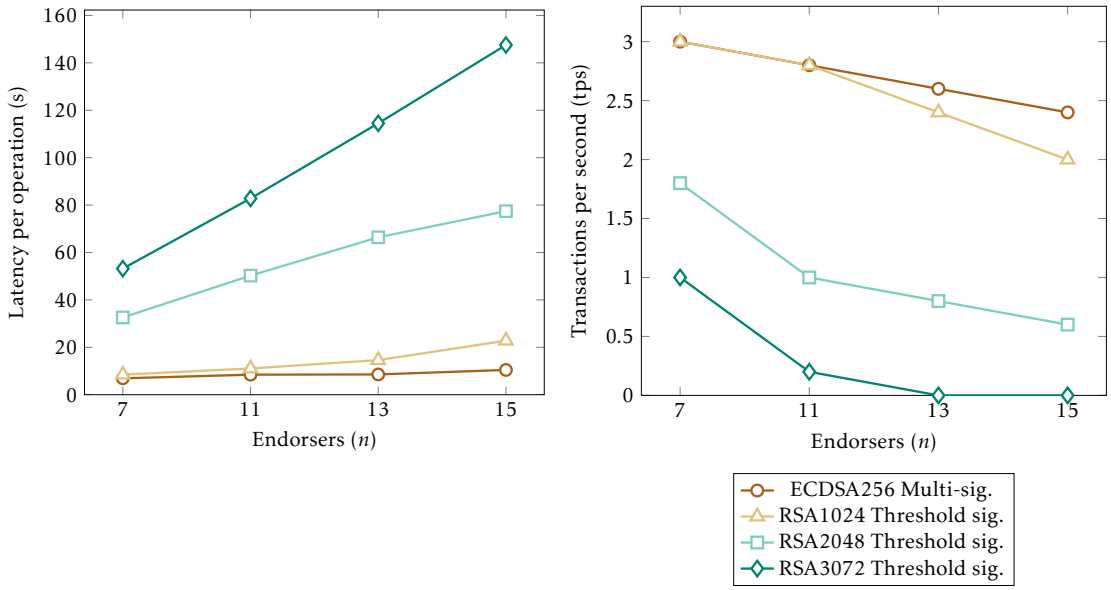

Figure 5.2: Prototype blockchain platform latency (left) and throughput (right) assessment with a varying number of endorsers, switching of signature schemes and varying RSA modulus for threshold signatures.

verification algorithm, the algorithm was able to verify transactions on the first generated combination of k shares. Most of the performance impact in threshold signatures seen here is believed to be related to the signature reconstruction process from the obtained signature shares during the signature verification algorithm, as the algorithm has to compute a polynomial interpolation of the k signature shares [67, 69] to generate a composite RSA signature before actually verifying it.

The results we obtained from this test were expected, and similar results were described by Stathakopoulou et al. [69]. While the authors obtained a higher throughput from signature generation and verification processes, their test-bench environment seems to hint that their experimental evaluation was conducted directly over the signature reconstruction and verification algorithms themselves as micro-benchmarks, in order to assess the feasibility of integrating them into HLF, rather than on a complete HLF deployment.

5.2.4 Impact of Transaction Block Sizes

In this benchmark we intended to evaluate the performance of our prototype blockchain platform with varying transaction block sizes. The size of transaction blocks in a blockchain system is an extensively discussed topic in the literature [20], especially for public blockchains such as Bitcoin, and is part of the blockchain scalability problem. Our interest in this topic is heightened by one of the advantages of threshold signatures over standard multi-signatures – signature size, a property that can influence the number of total transactions to be pushed into a single block.

So far, all tests have been done with block sizes of a maximum of 10 transactions. Block sizes in HLF are configured by three parameters: *i*) the number of maximum

transactions to enqueue into the block; *ii*) the maximum size of the block in Mb, which takes precedence over the former criteria; and *iii*) the preferred block size in Kb, which will be respected by the HLF when possible. Taking into account that we are sending transactions with very small applicational payloads, the block will reach the maximum number of transactions while retaining a size inferior or equal to the preferred block size. The majority of payload size in our tests are essentially the signatures appended to each transaction. We submitted the prototype blockchain platform in similar conditions to previous tests but for this benchmark we evaluated only ECDSA multi-signatures and RSA threshold signatures under comparable security levels (256 bits to 2048, respectively). We set the number of endorsers $n = 7$ and network size $l = 20$.

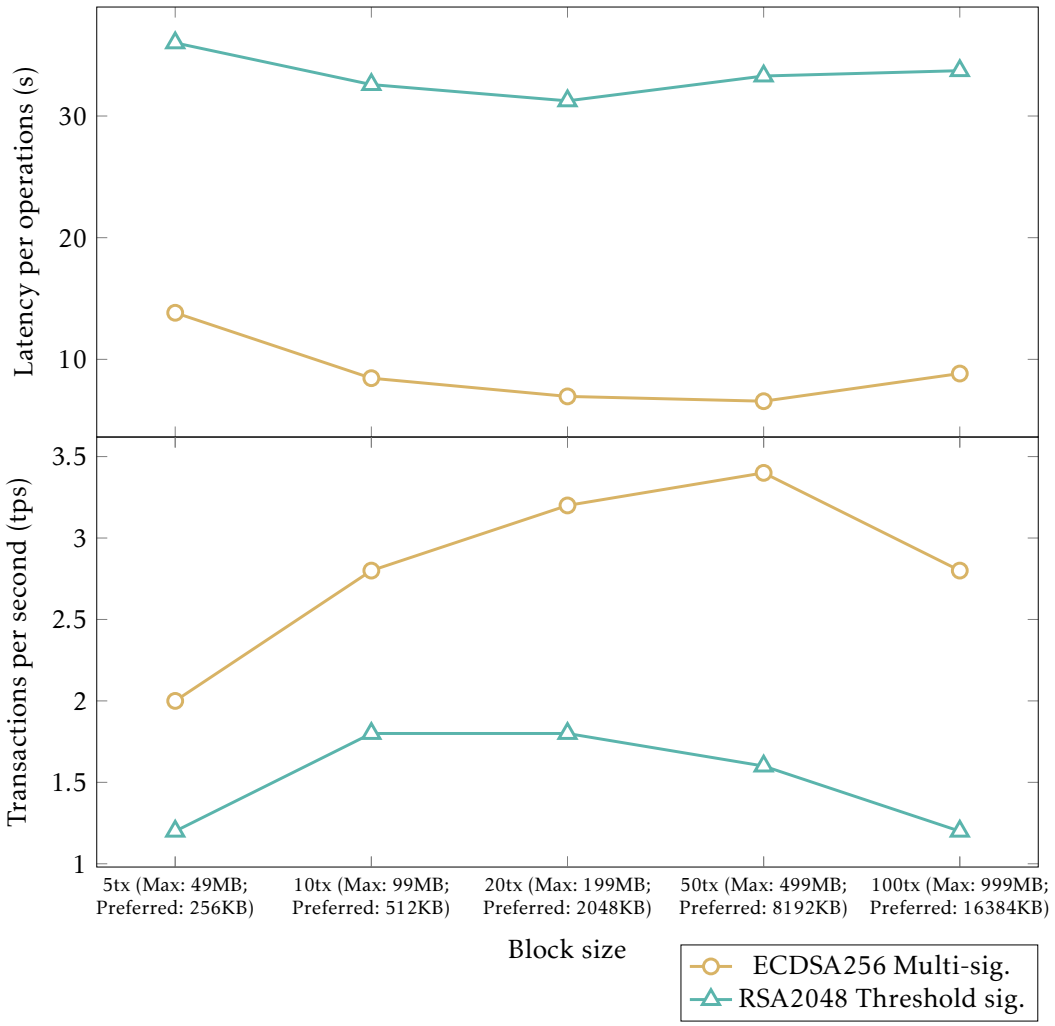

Figure 5.3: Prototype blockchain platform latency (top) and throughput (bottom) assessment with varying transaction block sizes.

Figure 5.3 illustrates our benchmark results in both latency and throughput. Let us focus on the throughput graph as more significant differences seem to exist between the *curves* of the two signature schemes. First of all, we notice that ECDSA multi-signatures

find their optimal throughput when block sizes are considerably large and near a maximum of 50 transactions, while for RSA threshold signatures, their optimal performance seems to be somewhere between the smaller 10 and 20 maximum transactions block sizes. This evidence is especially interesting for the context of using threshold signature versus multi-signature schemes. While a single ECDSA 256 signature is smaller in length than an RSA 2048 signature produced over the same message, the use of ECDSA multi-signatures results in a payload larger than a single threshold signature. In binary format, a single ECDSA 256 signature is 64 bytes in length, while for an RSA 2048 signature, the expected size is 256 bytes. A single transaction endorsed by a multi-signature ECDSA scheme, s being the ECDSA signature size, would have a maximum signature payload of $n \times s$. So, in this case $7 \times 64 = 448$ bytes when compared to the 256 bytes of an RSA threshold signature. The final size of the signature payload heavily influences the ordering service into deciding whether transactions all fit into a single block and can be readily dispatched or whether some transactions have to wait for the next block in order to be published due to block size restrictions. This influences the overall throughput of the HLF. Another observation we can make is that the slope in throughput in ECDSA multi-signatures from blocks with 50 transactions to blocks with 100 seems to be very accentuated in comparison with RSA threshold signatures. This leads us to think that there is a point in which the block size may grow to a number so elevated that the throughput of threshold signatures surpasses multi-signatures.

The results we see here were expected. One of the advantages of threshold signatures schemes is their fixed bounded size, while for multi-signatures the total size of the signature set varies with the number of signers. This is an important key factor of our dissertation, and shows that block sizes can be optimized for higher throughput while maintaining group-oriented decentralized trustability assumptions.

5.2.5 Fault-tolerance in Transaction Signature Schemes

The present benchmark assesses the fault-tolerance of different signature schemes harnessed by our prototype blockchain platform. In regular situations, a signature scheme that requires multi-signatures from a strict set of entities where a single party fails to sign a given message fails to verify. However, while Fabric implements a strict notion of which entities should sign a given transaction for a given contract, it also allows the configuration of what is called an *endorsement policy*, a user-defined logical expression that specifies the conditions in which a multi-signature set may be accepted or refused (e.g. party A may fail to sign message M as long as party B signs it instead). While this mechanism is an important component of HLF that leaves fault-tolerance conditions in signature verification up to organizations, we are more interested in the ability of using a signature scheme that may be able to represent the entities of a given endorsement policy as a group even if an individual did not sign it due to some kind of fault, hence threshold signatures.

So far, we have been using a *dummy* endorsement policy for Fabric accepting endorsements from any peer node and where the absence of peer signatures is allowed as long as at least one endorsement is produced, independently of the peer's organization. For this test, we stray away from this lax policy and define a policy that requires all n nodes to sign transaction proposals. For multi-signature schemes, this means that a peer fails to sign the transaction, the transaction itself fails to execute. We set the number of endorsers $n = 7$ and network size $l = 10$, and injected a payload of 100 transactions using Hyperledger Caliper, as in previous benchmarks. The reason why we set $l = 10$ in comparison with previous benchmarks is that a more significant output can be read in terms of throughput (e.g. in $l = 20$, we could have situations in which throughput would reach about zero transactions per second and it would be difficult for us to observe a tendency). Transaction block size was reset to a maximum of 10 transactions. As mentioned in the beginning of this section, the minimum quorum size for our threshold signature scheme is set at $k = f + 1$ where $n = 3f + 1$ are the number of endorsers needed to tolerate f faults. We execute this test in the presence of both crash and Byzantine faults by forcefully killing the container processes of peer nodes or by using tampered versions of the peer nodes image to output corrupted signatures, respectively. We increase the number of faults to $f = 1, 2, 3$ and trigger faults at time $t = 0$ (bear in mind that no node recovery mechanism exists).

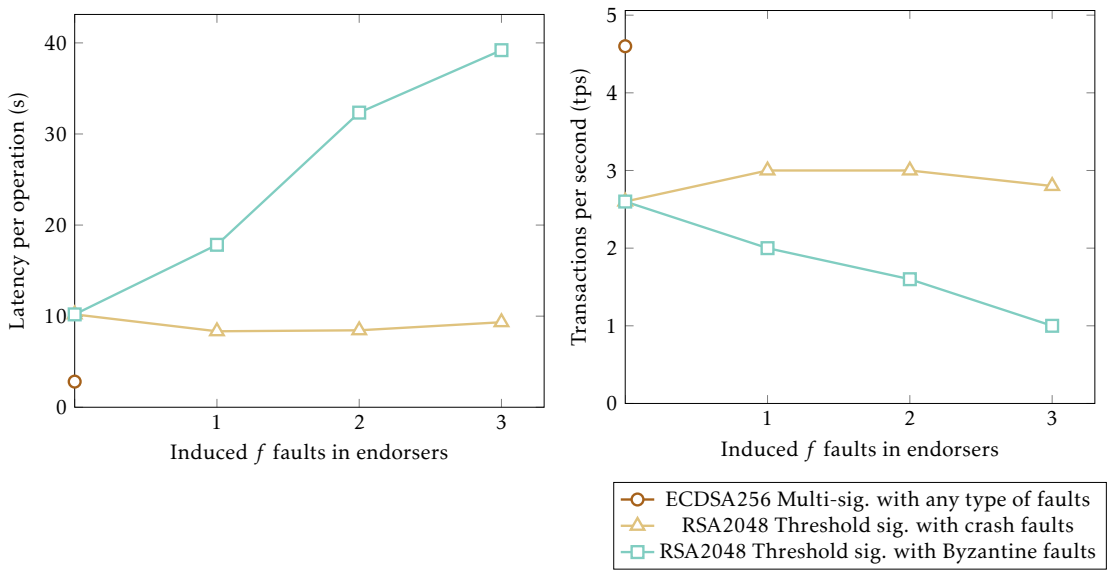

Figure 5.4: Prototype blockchain platform latency (left) and throughput (right) assessment in the presence of crash and Byzantine faults in endorser nodes with different signature schemes.

Figure 5.4 illustrates our results. First of all, notice that we were only able to evaluate ECDSA multi-signatures in conditions where no faults occur. After a single faulty endorsers, the endorsement policy triggered a failure in client and committer validations. Thus, no measurement was able to be made in $f = 1, 2, 3$. For threshold signatures, there

is an almost negligible throughput gain when crash faults start occurring. We believe this simply to be the fact that the client has to wait for less signatures before sending a transaction to the orderer nodes. Remember that all communication between the client and peer nodes is protected by gRPCS, so when a peer crashes, Caliper acknowledges the failure in a short span of time (since the TLS server-side socket closed) and no longer waits for responses from it. After a transaction block is published, less signature shares are combined for threshold signature verification by the committing peers, and confirming the transaction to the client should therefore be faster. For Byzantine failure conditions, an interesting spike in latency and loss of throughput is visible for threshold signatures. This occurs due to the optimistic signature verification algorithm we have implemented. When no faults occur, the algorithm is able to verify a threshold signature on the first attempt to reconstruct it from a single combination of k signature shares. By inducing faults, we can observe that the algorithm was unable to do so at a first try, and had to recompute a new combination for k out of the n received shares, reconstruct the composite signature and re-verify it. Each successive fault added the need to execute further combinations.

These results were expected and show the capability of threshold signatures to endure Byzantine participants or crashes in signature verification processes to a given threshold. We did not increase f further than 3 as this would naturally result in a failure to meet the threshold k. One thing to note is that our results in throughput seem to show linear decay of performance. However, our benchmark assesses a very small population of faulty nodes and a small number of endorsers. We believe this curve may reach exponential values when there is a very large number f of Byzantine parties, as stated in [69], since at the worst case scenario the algorithm may have to compute all combinations of signature shares $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ to either find out no combination of k will result in a valid signature or that the last combination is the only valid one.

5.2.6 Smart Hub Scalability and Throughput

For this benchmark and the following, we deviate our focus of the prototype blockchain platform to the outermost layer of our system and look to the edge environment. Specifically for this benchmark, the target of our evaluation was our prototype smart hub API. Here, we evaluate the smart hub API from an HTTP/CoAP client in an truly E2E (end-to-end) scenario, as we used our prototype blockchain platform as supporting asset to the smart hub. Thus, latency values presented here are influenced by the blockchain platform conditions we assessed in previous benchmarks.

For this test we used our HTTP/CoAP client implementation, which is capable of launching multiple client threads simultaneously, each producing a workload of 10 operations (60% read operations and 40% write operations). For simplicity, we assess this benchmark in latency per operation requested to the hub only. We varied the number of clients c to $c = 1, 10, 25, 75$ to assess the capability of our smart hub implementation

to scale in an emulated yet ever increasing environment of IoT devices and users. We ran the smart hub in its *emulated* version, i.e. executing it locally from our test laptop. The smart hub CoAP and HTTP thread pools for attending requests were both limited to 12 threads. Regarding the signature schemes of the supporting blockchain platform, we set it to use only ECDSA multi-signatures of 256 bits and RSA threshold signatures of 2048 bits. HTTP communication channels between the client and the hub were protected with TLS 1.2, while for CoAP no security protocol was leveraged for in transit traffic[6]. It is worth to mention that the CoAP protocol could be implemented using TCP and even configured to use TLS, but our version communicates solely through UDP.

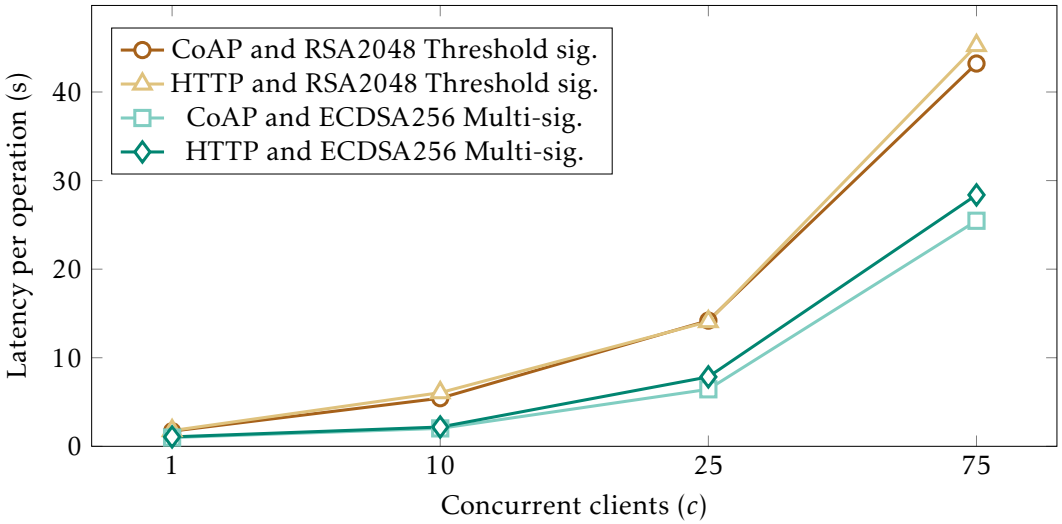

Figure 5.5: Prototype smart hub scalability assessment with CoAP and HTTP communication protocols and different supporting blockchain transactions signature schemes.

Figure 5.5 illustrates our benchmark results. First, let us notice that when using the ECDSA multi-signature scheme, the observed latency is lower than when using RSA threshold signatures, a tendency we had already identified in previous assessments. The second observation we can make is that under small to medium concurrent client conditions ($1 \leq c \leq 25$) our implementation scales reasonably well. We foresee that our implementation would be adequate for small IoT enviroments that do not require immediate real-time response rates, such as a smart home or a smart bicycle rack where it is also unlikely to have an extreme amount of IoT devices connected to a single hub. Again, bear in mind that the supporting blockchain platform is not running on what we would call a production environment; it is running on single virtualized machine and much lower latency values could possibly be obtained from the smart hub if a more powerful supporting blockchain infrastructure was available. When $c = 75$, the latency per operation increases considerably, as the smart hub has to enqueue multiple requests in parallel while waiting on previous transactions to the blockchain to complete. As there are limits

[6]As of the writing of this document, the *mbed* CoAP library did not support DTLS, although we did implement DTLS in an alternative version of the smart hub.

on the CoAP and HTTP server-side thread pools attending requests, as not to consume too much resources on the host machine, there is a bottleneck at the smart hub communication entry point. The third and final observation is the difference in latency between the CoAP and HTTP protocols. CoAP communication seems to be able to achieve lower latency values by marginal differences.

While a higher difference in latency between the CoAP and the HTTP protocols could be expected, taking into account that we use TLS for HTTP and that CoAP is UDP datagram-oriented (and thus, there should be a speed advantage of UDP over TCP as the sender does not have to establish a TCP connection and wait for an acknowledge packet before sending the actual message), we must take into account that CoAP requires an acknowledge packet for every datagram it sends, while TCP, after its slow-start phase, is able to acknowledge multiple packets with a single ACK message [51]. The CoAP specification [66] explicitly states that the protocol provides reliability in messages marked as confirmable through acknowledge, reset and non-confirmable return messages. However, the biggest benefit of CoAP is related to resource consumption, especially for client devices, rather than speed. This is something that will be evident in the next section.

However, before we proceed to the next section, let us first evaluate the performance of the smart hub API on a physical device with less resources than our local laptop: a Raspberry Pi, an equipment which would feasibly be seen in an IoT environment and that could possibly assume the role of a lightweight smart hub deployment. For this test we benchmarked the physical smart hub against the emulated one on similar conditions. The number of clients was set to $c = 10$ and the same workload and testing conditions aforementioned were replicated except for signature schemes, which we restricted to RSA threshold signatures of 2048 bits only.

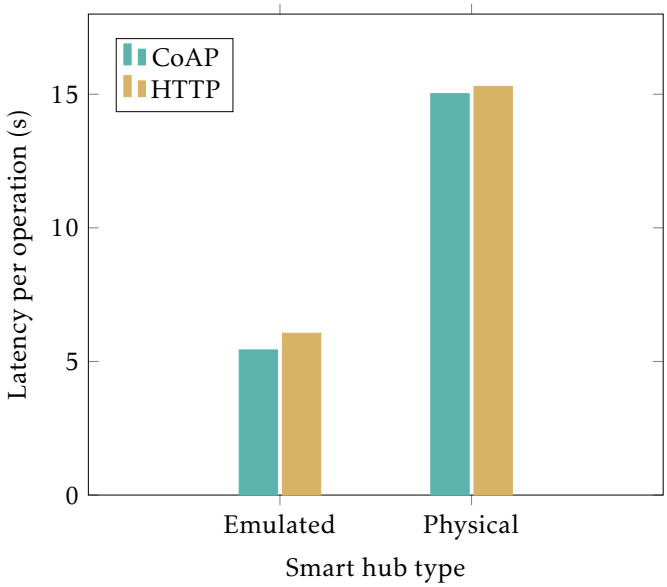

Figure 5.6: Performance comparison between an emulated version of the smart hub prototype and a physical version deployed on the Raspberry Pi.

The results of this test are shown in Figure 5.6. Latency per operation on the physical smart hub was quite higher than our laptop, which was expected given the specifications of our laptop versus the specifications of a Raspberry Pi. In terms of CoAP versus HTTP performance, CoAP was again marginally better in latency. This test lets us partially assess our host machine requirements for the smart hub API having the Raspberry Pi as a minimum baseline. For small IoT environments where waiting a considerable amount of seconds for an operation to execute, a Raspberry Pi would be an acceptable host. We foresee more complex smart hubs to be somewhere in between our emulated deployment and the physical one.

5.2.7 Smart Hub and Client Resource Consumption

For this benchmark, our intention was to assess the consumption of hardware resources at the edge when IoT devices and other client applications exchange data with the smart hub. Our metrics are in allocated memory (in Mb) and CPU usage (%) by the respective applications.

To conduct this test, we set the supporting prototype blockchain platform to only use RSA threshold signatures with 2048 bits. We then subjected the smart hub to a varying number of concurrent clients, each producing a workload of 10 operations (60% read operations and 40% write operations), similarly to the conditions of the previous benchmark, and ran the smart hub in its emulated form. During the test we simultaneously monitored both the smart hub and the client issuing HTTP/CoAP requests and analyzed their runtime heap dumps using the tools described in the beginning of this chapter.

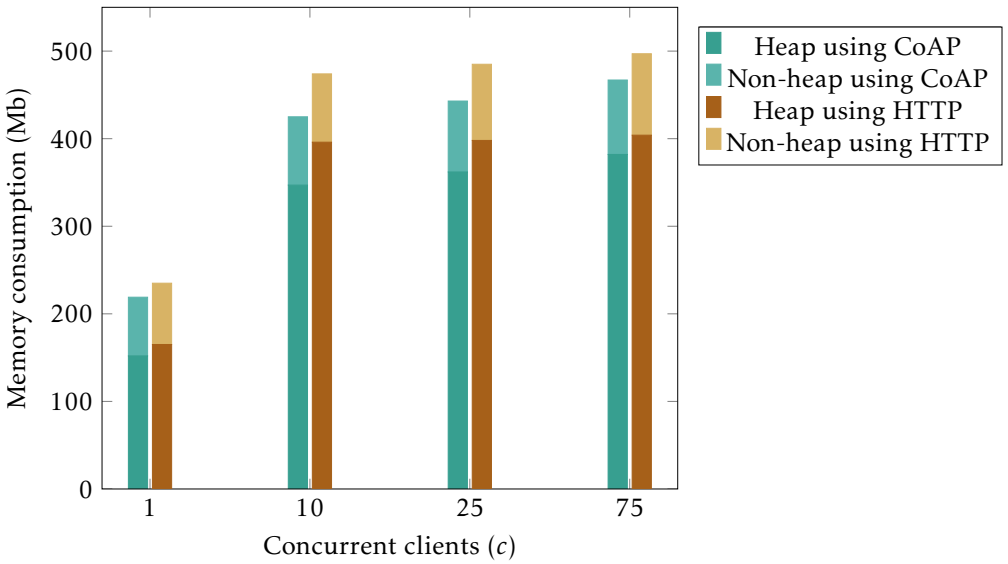

Figure 5.7: Peak memory consumption of the smart hub prototype with CoAP and HTTP communication protocols.

Figure 5.7 shows our results in terms of peak allocated memory on the smart hub. We divide results in heap memory and non-heap memory (stack, data and code segments,

etc.) and show the peak memory usage before JAVA garbage collection occurred. Non-heap memory is relevant for us to understand the memory footprint of the execution of the smart hub threads. Heap memory, on the other hand, is what truly allows us to assess the dynamic allocation of data that may result from requesting operations to the hub. In our results, we again compare both protocols: CoAP and HTTP. However, it is critical to note that the memory results seen here correspond to the whole smart hub application as a unit. Thus, while difference in memory footprints for CoAP and HTTP may be visible, a big percentage of memory is consumed by the smart hub to execute other inherent functions, such as orchestrating communication with blockchain broker nodes, storing smart contract cache, maintaining a database connection, and so on. Looking at our result set, we can see some small differences in memory consumption. In general, we can see that CoAP and HTTP increase their overall memory footprint significantly when the number of c clients increase to a concurrency scenario. However, in comparison with one another, their non-heap memory consumption was nearly identical. The big difference between both lies in the consumption of heap memory, where we can see a discernible contrast. When $c = 10$ we can see that the gap in heap memory reaches the order of magnitudes of nearly 50Mb between CoAP and HTTP. Afterwards, when $c = 75$, the heap footprint between both protocols seems to converge. Overall, the average consumption of the smart hub when using both protocols is around $m_1 \approx 311$Mb for CoAP and $m_2 \approx 341$Mb for HTTP. Difference in heap memory consumption between both is $\Delta m = 30$Mb $\approx 9\%$. Bear this value in mind, as we proceed to evaluate the CPU usage of the smart hub and the resource footprint of the client application.

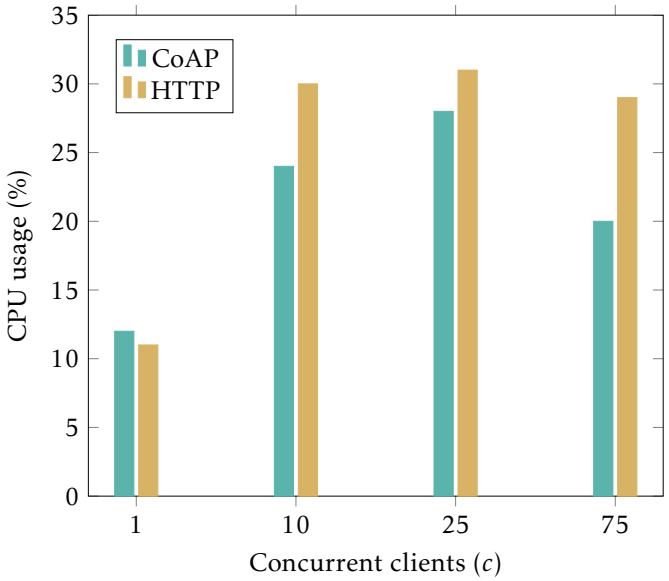

Figure 5.8: Peak CPU usage of the smart hub prototype with CoAP and HTTP communication protocols.

Figure 5.8 shows the peak CPU usage of the smart hub during our assessment. For a single client ($c = 1$), the smart hub consumes more CPU amid CoAP communication

that in does for HTTP. Then, when the number of concurrent clients reaches values of $c = 10, 25, 75$, we can then observe that HTTP CPU usage surpasses CoAP. In average, CoAP reaches a CPU usage percentage on the smart hub of $c_1 = 21\%$ and HTTP reaches around $c_2 = 25\%$, with a mere difference of $\Delta c = 4\%$ between both. Let us now assess the results of the client evaluation and discuss upon both results.

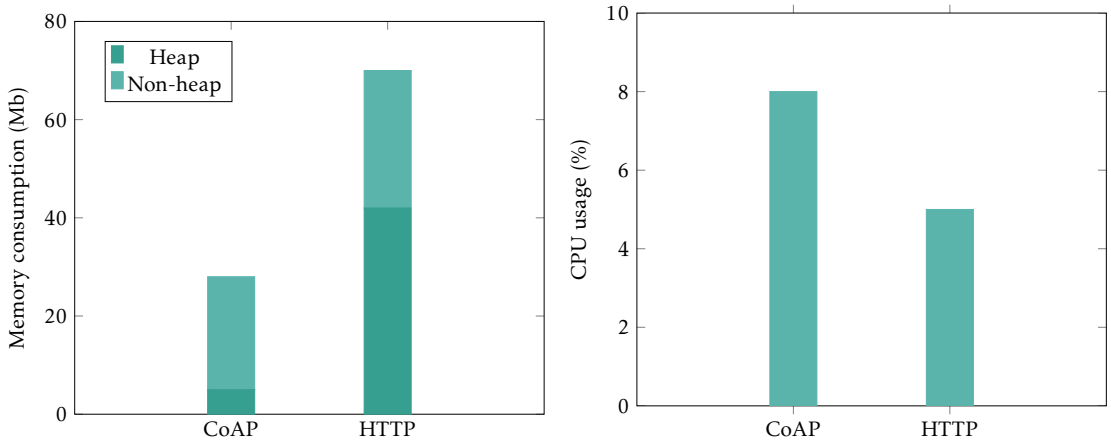

Figure 5.9: Peak memory consumption (left) and CPU usage (right) evaluation of the benchmarking client implementation using CoAP and HTTP communication protocols.

Figure 5.9 illustrates the results for CoAP and HTTP communication of monitoring our client implementation in the previous benchmark when a single thread is in execution. This benchmark is especially important as it assesses the impact of communication with our prototype for what could be an resource-constricted IoT device. Looking at our results, we can observe that, as in previous benchmarks and as expected, non-heap memory is roughly the same between HTTP and CoAP. However, we see a completely different order of magnitude related to the consumption of heap memory between both protocols, from just $m_1 = 5$Mb in CoAP to $m_2 = 42$Mb in HTTP. Remember that in the smart hub we were looking at an 9% decrease, while here we are reaching around 88%. In an IoT device where memory hardware is optimized to the minimum as viably possible in relation with the cost of production of the device, a small memory footprint is essential. This is exactly what our test shows: CoAP is much more capable of saving memory for client applications than HTTP is.

The contrast in memory consumption for both the smart hub and the client is expected. When comparing CoAP and HTTP, CoAP sends messages in what it defines as a compressed format [66]. A UDP packet has a minimum header of 8 bytes and a CoAP header is merely 4 bytes long. In comparison, a HTTP header completely overshadows CoAP, as a TCP header alone has a limit of 20 bytes while HTTP can be quite complex and thus has no limit over the header size [51]. Curiously, the CPU load for the client when issuing CoAP requests is higher than when using HTTP (the same situation occurs on the smart hub when a single client communicates with the hub). This can be a drawback for IoT devices which are limited in battery resources. A higher CPU load can lead to a faster

consumption of battery and reduce the reliability of the device. While we did not explore the root cause for this increase, we suspect it may be an implementation characteristic of the *mbed* CoAP library due to a possible lack of resource optimization.

5.3 Summary

In this chapter we described our experimental assessment results over our implemented prototype. We conducted an experimental evaluation on both the blockchain platform component of the prototype as well as the smart hub API in an attempt to obtain the a set of criteria that could answer the questions mentioned in the beginning of this chapter. A rigorous analysis was executed for each benchmark section, which are succinctly summarized below:

- The baseline expected performance of running the Hyperledger Fabric on a single cloud server in a virtualized Docker environment;
- The baseline expected performance of the BFT consensus ordering service in the same conditions and its elasticity when the number of replicas increases;
- The average latency and throughput of the original blockchain platform with an increasing network topology and a higher number of transaction endorsing nodes;
- The average latency and throughput of our prototype blockchain platform with a varying number of endorsers and with the usage of different transaction signature verification schemes: ECDSA multi-signatures and RSA threshold signatures (with different RSA modulus values);
- The impact and possible performance optimizations that can be performed with changing ledger block sizes to fit more or less transactions into a single block, taking into account the aforementioned signature schemes;
- The capability for withstanding Byzantine and crash faults in endorsing processes for both signature schemes;
- The scalability and throughput of our smart hub implementation in the presence of an ever increasing number of clients that can be seen as an approximation to IoT things in a real IoT environment, alongside a technical assessment of the possible advantages of harnessing the CoAP protocol over HTTP for such devices;
- The resource consumption, in terms of memory and CPU usage, for both smart hub and benchmarking client implementations when using CoAP or HTTP;

Answering our own questions, our conclusion is the following. Overall, given our results, we believe that on a physically distributed cloud infrastructure (preferably heterogeneous in cloud providers), our prototype blockchain platform can be deployed to

achieve high throughput with low latency values, even when using RSA threshold signature schemes. As seen in our assessments, our scheme outperforms multi-signature schemes in conditions where there is a strict endorsement policy for transactions and some of endorsing entities may be faulty, thus providing a truly decentralized approach for transaction endorsement and verification. Additionally, we were able to achieve optimal throughput for threshold signatures in smaller block sizes when compared to multi-signatures, an evidence that proves our system is capable of producing signature payloads with much smaller lengths. As RSA threshold signatures of 3072 bits cause some overhead on the system, our recommendation to keep the prototype on pair with current security practices is to use keys of 2048 bits.

At the edge, we were able to provide an implementation that scales relatively well for small to medium IoT environments. In large IoT environments where large numbers of IoT devices communicate simultaneously, our proposition would be to leverage further hubs to hierarchically aggregate communication. We were also able to leverage a protocol whose memory footprint is more adequate for the IoT device range when compared to popular network protocols such as HTTP. Regarding CPU usage, we obtained some results that were unexpected but we assume this may be due to some lack of optimization in the library we leveraged.

As a final remark, we believe that our blockchain-supported prototype is a valid proof-of-work for a scalable and more decentralized cloud-enabled blockchain IoT architecture, which can be an alternative to traditional IoT architectures.ro

Chapter 6

Conclusion and Final Remarks

In this chapter we conclude the present document with some conclusions regarding the research that was conducted, our system model and its prototype implementation, what was done and some of the limitations that exist. We will also present some suggestions that would allow someone to potentially address these limitations in the future.

6.1 Conclusions

The research conducted during this thesis allowed us to grasp the current neglected state of the Internet-of-Things (IoT), security and privacy-wise, and explore on the cloud-first architectures on which this paradigm currently sits upon. We studied on the inner workings of blockchain technology and existing blockchain platforms, whose properties seem to meet with the IoT's needs regarding robustness, auditability and security. To understand the limitations and advantages of existing blockchain platforms, an analysis on software engineering, system architecture and programming support was done for a representative group of platforms, which we provide as a contribution to consolidating the current state-of-the-art of blockchain technology. We also reviewed existing decentralized ledgering solutions for the IoT and scrutinize the use of threshold cryptography for blockchain systems as an approach for integrating group-oriented trustability assumptions into blockchain architecture.

The system model we propose for IoT environments presents itself as a tiered blockchain-based Byzantine fault-tolerant model ensuring decentralization of trust and non-repudiation capabilities while enabling independent auditability of data for every involved party, that can be heterogeneously deployed across cloud providers to achieve scalability and prevent vendor lock-in. Our intention with this model is to improve on current IoT architectures as a first step to taking back control from service providers and making sure every party

is protected from fraudulent behaviours, while abstracting the weight of the blockchain protocol from IoT devices, of which most are heavily resource-constrained.

Based on our model, we built a fully functional end-to-end prototype, and thus implemented and extended a system on top of a known blockchain platform we evaluated thoroughly in our related work – the Hyperledger Fabric – adding the services imposed by our system model. Smart hub and client implementations were also built and used to mediate communication with the blockchain back-end and to issue new transactions and queries to the hub, respectively. Our implementation was done having into account the resource restrictions of most IoT devices, harnessing both CoAP and HTTP communication protocols.

The executed experimental evaluation allowed us to assess how our prototype performed in terms of throughput and latency with a BFT consensus mechanism, varying signature schemes (multi-signatures and threshold signatures), algorithms and key sizes, in the presence of crash faults and Byzantine behaviour, with a varying size for transaction blocks, and with an increased number of clients and traffic pressure on smart hubs, as to infer its scalability. From the results we conclude that our prototype is valid proof-of-work for a scalable and more decentralized cloud-enabled blockchain IoT architecture that is capable to address the drawbacks of today's cloud-first IoT architectures.

Overall, we addressed the problems proposed to us in this thesis and devised a model and proof-of-work of that same model, capable of providing itself as an alternative to traditional IoT architectures, in which the nature of the service is still cloud-centric while allowing independent scrutiny by end users and better control of their own data, devices and transactions, with the added resiliency properties of a blockchain together with decentralized and robust signature verification processes, thereby preventing tampering and corruption of data by a malicious party.

6.2 Future Work

We were able to develop a functional prototype of our system model, although there are still some issues that could be improved upon in the future.

Regarding our blockchain services implementation, our solution for leveraging threshold signatures for transaction signing and verification may present a few performance drawbacks related to having to delegate signature construction and verification from blockchain nodes main execution process to an external dedicated component. Albeit its modularity, this implementation requires the integration of server-side code for establishing connections, accepting requests and maintaining thread pools for attending tasks. As stated earlier, this was mostly due to lack of cryptographic primitives for supporting threshold signatures in Golang, the language in which Hyperledger Fabric is implemented. A direction for future work would be further investigation of a solution implementing threshold signatures directly in Golang and remeasuring the performance of using RSA threshold signature schemes with various key sizes versus standard ECDSA

multi-signatures, in order to assess if considerable performance gain is achievable in return for the modularity of this component.

Again in our implementation of threshold signing and verification processes, we inject key material directly into blockchain peers. Our solution, however, is compatible with the notion of a trusted dealer. A direction for future work could be implementing such a notion in a way that does not bring about centralization of the generation key material to a single entity, since this presents a single point of failure (e.g. the private keys could be exposed, corrupted keys could be distributed, etc.). A solution could be to either make use of a consortium consensus service to generate cryptographic material and thus act as a trusted dealer. Another direction could be studying and implementing the generation of threshold signature keys in an interactive algorithm between participating blockchain peers rather than depending on a trusted dealer.

Related to our smart hub component, other relevant research directions are the support of other interoperability protocols that are designed for implementation in IoT devices. This effort must be seen as a natural evolution and extension of smart hub modules, in convergence with the broader view expressed in the discussion of the functionality of smart hubs for IoT edge-based environments. A possible initial approach for this could consist in leveraging the MQTT protocol for the smart hub API in order to communicate with resource-limited IoT devices, alongside CoAP. While both are recognized open standards for IoT communication, MQTT is more oriented to publish-subscribe architectures with many-to-many communication flows in which there can be a single broker publishing information to a set of subscribers, while CoAP works in a traditional client-server one-to-one communication flow. While CoAP may be sufficient for simpler IoT environment scenarios, we find that MQTT may better adapt to the nature of smart hubs that require sharing data between several aggregated devices.

There is a space to research on better expressiveness conditions of smart contracts to support IoT operations enabled by blockchain transactions and expressed smart contracts. Smart contracts can be used to define parameters, rules and invariants for different levels of execution requirements. These requirements range from specific IoT application-level validation guarantees, to parameters and conditions dynamically regulating the internal services of blockchain service planes (including storage, aggregation of transactions, management of stored blocks, ordering semantics, or endorsement criteria), in a more re-configurable or possibly dynamic re-configurable runtime environment. In terms of smart contract verification and persistency at the Contract Interpreter component of our prototype, improvements can be made as future work so that the smart hub may be able to update locally cached contracts upon new versions being deployed to the blockchain. Storing some sort of meta-data and version numbers for each contract, alongside a mechanism to ensure consistency between the blockchain services and the smart hub could be a direction for addressing this improvement.

Going back on our system model, our extended interaction model was not addressed in our prototype. Due to time restrictions, we only implemented the base interaction

model. A direction for future work would be following on the extended interaction model to implement a similar set of services to those installed on the smart hub implementation in user devices capable of enough computation and with enough resources (e.g. smartphones) to communicate directly with blockchain broker nodes. A suggestion to address this as future work could be in extending our work for the developed Android application so that it does not rely solely on a smart hub.

Our testing environment also had some limitations in terms of computation and resource capabilities that prevented us from running tests with blockchain networks having a high number of nodes. As stated earlier, we ran a virtualized Docker network over a single server. However, we were unable to produce high levels of throughput without resource limitations of the machine severely impacting our tests. Thus, we were unable to explore the total throughput potential of the Hyperledger Fabric and of our prototype. Thus, we believe that the approach for properly testing a highly dense blockchain network, and something that could be assessed in future work, would be to run our solution in a physically distributed environment.

In some specific benchmarks, such as measuring the throughput of the original Hyperledger Fabric platform and in measuring the resource consumption of the CoAP protocol versus HTTP, we obtained some peculiar results. For the former, we believe that Fabric's performance decays exponentially due to the gossip protocol, which we suspect to have a $O(n^2)$ temporal complexity in the number of peers of the blockchain network. However, we believe further testing has to be executed in future work to confirm this hypothesis, as our results seem to show a stabilizing tendency in throughput when network density increases. For the latter, while we obtained expected results in terms of memory consumption between both protocols, we also observed that CPU load of CoAP was higher than HTTP for client implementations, something which we assume to be a lack of resource optimization in the CoAP library we use for our prototype. Further analysis can be done in future work to identify the root cause of this issue.

Another issue that was not dealt with in this thesis, is the future comparison of the obtained results with the officially supported BFT algorithm that will be launched for the HLF consensus plane in the near future, using more extensive benchmarks.

Finally, there is another research direction in other aspects with relevant impact for scalability purposes. We summarize some of these research directions in two different groups of concerns: scale-in and scale-out concerns. For scale-in concerns we must observe the on-going research proposals in addressing better performance figures for HLF (and other permissioned blockchains in the ongoing research agenda), as well in the approach of separation concerns decoupling membership services, consensus planes for consistency control and decentralized hierarchies of blockchains, for example in a tree-based architectural model or by using sharding models interconnecting different blockchain domains. For scale-out purposes, we can extend the current system model in the dissertation by interconnecting smart hubs in edged-based blockchains, or possibly composed by other smart hubs in upper-level hierarchies.

Bibliography

[1] I. Allison. *Ethereum-based Slock.it reveals first ever lock opened with money*. Ed. by I. B. Times. 2015. URL: http://www.ibtimes.co.uk/ethereum-based-slock-reveals-first-ever-lock-opened-money-1527014 (visited on 12/27/2017).

[2] *Amazon Echo*. URL: https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-Alexa-Black/dp/B00X4WHP5E (visited on 02/03/2018).

[3] *Apple HomeKit*. URL: https://developer.apple.com/homekit (visited on 02/03/2018).

[4] M. Banerjee, J. Lee, and K.-K. R. Choo. "A blockchain future to Internet of Things security: A position paper." In: *Digital Communications and Networks* (2017). URL: http://www.sciencedirect.com/science/article/pii/S2352864817302900 (visited on 01/07/2018).

[5] E. Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin." In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. URL: http://ieeexplore.ieee.org/document/6956581/ (visited on 12/27/2017).

[6] A. Bessani, J. Sousa, and M. Vukolić. "A Byzantine Fault-tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform." In: *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '17. ACM, 2017, 6:1–6:2. URL: http://doi.acm.org/10.1145/3152824.3152830 (visited on 08/07/2018).

[7] *BFT ordering service for Hyperledger Fabric*. URL: https://github.com/bft-smart/fabric-orderingservice (visited on 08/07/2018).

[8] D. Boneh, R. Gennaro, and S. Goldfeder. *Using Level-1 Homomorphic Encryption To Improve Threshold DSA Signatures For Bitcoin Wallet Security*. 2017. URL: http://www.cs.haifa.ac.il/~orrd/LC17/paper72.pdf (visited on 09/21/2018).

[9] D. Boneh, M. Drijvers, and G. Neven. "Compact Multi-Signatures for Smaller Blockchains." In: *IACR Cryptology ePrint Archive* 2018 (2018). URL: https://eprint.iacr.org/2018/483.pdf (visited on 09/21/2018).

[10] F. Bonomi et al. "Fog Computing and Its Role in the Internet of Things." In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. URL: http://doi.acm.org/10.1145/2342509.2342513 (visited on 02/02/2018).

[11] C. Boyd. "Digital multisignatures." In: *Cryptography and Coding*. Clarendon Press, Oxford, 1986, pp. 241–246.

[12] V. Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. URL: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 12/27/2017).

[13] C. Cachin. "Distributing Trust on the Internet." In: *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*. DSN '01. IEEE Computer Society, 2001, pp. 183–192. ISBN: 0-7695-1101-5. URL: http://dl.acm.org/citation.cfm?id=647882.738080 (visited on 12/23/2017).

[14] C. Cachin and A. Samar. "Secure distributed DNS." In: *International Conference on Dependable Systems and Networks, 2004*. 2004, pp. 423–432.

[15] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance." In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf (visited on 12/27/2017).

[16] K. Christidis and M. Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things." In: *IEEE Access* 4 (2016), pp. 2292–2303. URL: http://ieeexplore.ieee.org/document/7467408/ (visited on 12/27/2017).

[17] A. Clemens. *IoT Security: Does Such a Thing Exist?* 2017. URL: https://www.tripwire.com/state-of-security/security-data-protection/iot/iot-security-does-such-a-thing-exist/ (visited on 01/11/2018).

[18] G. Coulouris, J. Dollimore, and T. Kindberg. In: *Distributed Systems: Concepts and Design*. 3rd. Addison-Wesley, 2001, pp. 451–462. ISBN: 9780201619188.

[19] *Counterparty*. URL: https://counterparty.io/ (visited on 12/27/2017).

[20] K. Croman et al. "On Scaling Decentralized Blockchains - (A Position Paper)." In: *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*. 2016, pp. 106–125. URL: https://doi.org/10.1007/978-3-662-53357-4_8 (visited on 12/27/2017).

[21] Y. Desmedt. "Society and Group Oriented Cryptography: a New Concept." In: *Advances in Cryptology — CRYPTO '87: Proceedings*. Springer Berlin Heidelberg, 1988, pp. 120–127. ISBN: 978-3-540-48184-3. URL: https://doi.org/10.1007/3-540-48184-2_8 (visited on 12/23/2017).

[22] Y. Desmedt and Y. Frankel. "Threshold cryptosystems." In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Springer New York, 1990, pp. 307–315. ISBN: 978-0-387-34805-6. URL: https://doi.org/10.1007/0-387-34805-0_28 (visited on 12/23/2017).

[23] Y. Desmedt and Y. Frankel. "Shared generation of authenticators and signatures." In: *Advances in Cryptology — CRYPTO '91: Proceedings*. Springer Berlin Heidelberg, 1992, pp. 457–469. ISBN: 978-3-540-46766-3. URL: https://doi.org/10.1007/3-540-46766-1_37 (visited on 12/23/2017).

[24] A. Dorri, S. S. Kanhere, and R. Jurdak. "Towards an Optimized BlockChain for IoT." In: *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. IoTDI '17. ACM, 2017, pp. 173–178. ISBN: 978-1-4503-4966-6. URL: http://doi.acm.org/10.1145/3054977.3055003 (visited on 01/07/2018).

[25] *EtherTweet*. URL: http://ethertweet.net/ (visited on 12/27/2017).

[26] L. Fair. *What Vizio was doing behind the TV screen*. 2017. URL: https://www.ftc.gov/news-events/blogs/business-blog/2017/02/what-vizio-was-doing-behind-tv-screen (visited on 02/03/2018).

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process*. 1985. URL: https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf (visited on 12/27/2017).

[28] T. L. Foundation. *The Hyperledger project*. URL: https://www.hyperledger.org/ (visited on 12/22/2017).

[29] Y. Frankel et al. "Proactive RSA." In: *Advances in Cryptology — CRYPTO '97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings*. Springer Berlin Heidelberg, 1997, pp. 440–454. ISBN: 978-3-540-69528-8. URL: https://doi.org/10.1007/BFb0052254 (visited on 12/23/2017).

[30] R. Gennaro et al. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems." In: *Advances in Cryptology — EUROCRYPT '99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings*. Springer Berlin Heidelberg, 1999, pp. 295–310. ISBN: 978-3-540-48910-8. URL: https://doi.org/10.1007/3-540-48910-X_21 (visited on 12/23/2017).

[31] S. Gibbs. *Hackers can hijack Wi-Fi Hello Barbie to spy on your children*. 2015. URL: https://www.theguardian.com/technology/2015/nov/26/hackers-can-hijack-wi-fi-hello-barbie-to-spy-on-your-children (visited on 02/03/2018).

[32] S. Goldfeder et al. *Securing Bitcoin Wallets via Threshold Signatures*. Tech. rep. University of Princeton, 2017. URL: http://www.cs.princeton.edu/~stevenag/bitcoinnthresholdsignatures.pdf (visited on 12/23/2017).

[33] G. Greenspan. *MultiChain Private Blockchain – White Paper*. 2015. URL: https://www.multichain.com/download/MultiChain-White-Paper.pdf (visited on 12/27/2017).

[34] L. Harn. "Group-oriented (t, n) threshold digital signature scheme and digital multisignature." In: *IEE Proceedings - Computers and Digital Techniques* 141.5 (1994), pp. 307–313.

[35] E. Heilman, F. Baldimtsi, and S. Goldberg. "Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions." In: *IACR Cryptology ePrint Archive*. 2016. URL: https://eprint.iacr.org/2016/056.pdf (visited on 12/27/2017).

[36] A. Herzberg et al. "Proactive Secret Sharing Or: How to Cope With Perpetual Leakage." In: *Advances in Cryptology — CRYPT0' 95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings*. Ed. by D. Coppersmith. Springer Berlin Heidelberg, 1995, pp. 339–352. ISBN: 978-3-540-44750-4. URL: https://doi.org/10.1007/3-540-44750-4_27 (visited on 01/07/2018).

[37] A. Herzberg et al. "Proactive Public Key and Signature Systems." In: *Proceedings of the 4th ACM Conference on Computer and Communications Security*. CCS '97. ACM, 1997, pp. 100–110. ISBN: 0-89791-912-2. URL: http://doi.acm.org/10.1145/266420.266442 (visited on 01/07/2018).

[38] K. Hill. *When 'Smart Homes' Get Hacked: I Haunted A Complete Stranger's House Via The Internet*. 2013. URL: https://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack/#43ac0cb7e426 (visited on 02/03/2018).

[39] P. N. Howard. *How big is the internet of things and how big will it get?* Tech. rep. The Brookings Institution, 2015. URL: https://www.brookings.edu/blog/techtank/2015/06/08/how-big-is-the-internet-of-things-and-how-big-will-it-get/ (visited on 02/03/2018).

[40] *Hydrachain*. URL: https://github.com/HydraChain/hydrachain (visited on 12/27/2017).

[41] *Hyperledger Burrow*. URL: https://github.com/hyperledger/burrow (visited on 12/27/2017).

[42] *Hyperledger Fabric*. URL: https://github.com/hyperledger/fabric (visited on 12/27/2017).

[43] *Hyperledger Iroha*. URL: https://github.com/hyperledger/iroha (visited on 12/27/2017).

[44] *Hyperledger Sawtooth*. URL: https://github.com/hyperledger/sawtooth-core (visited on 12/27/2017).

[45] M. Iansiti and K. R. Lakhani. "The Truth About Blockchain." In: *Harvard Business Review* (January–February 2017), pp. 118–127. URL: https://hbr.org/2017/01/the-truth-about-blockchain (visited on 12/27/2017).

[46] IBM. *Maersk and IBM Unveil First Industry-Wide Cross-Border Supply Chain Solution on Blockchain*. 2017. URL: http://www-03.ibm.com/press/us/en/pressrelease/51712.wss (visited on 12/27/2017).

[47] *Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. Standard. International Organization for Standardization, 2016.

[48] *IoT Standards and Protocols*. URL: https://www.postscapes.com/internet-of-things-protocols/ (visited on 09/21/2018).

[49] A. Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts." In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 839–858. URL: http://ieeexplore.ieee.org/document/7546538/ (visited on 12/27/2017).

[50] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem." In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/ (visited on 12/27/2017).

[51] M. Lanter. "Scalability for IoT Cloud Services." Master's thesis. ETH Zurich, 2013. URL: http://www.lantersoft.ch/blog/scalability/MA_lanter_martin_2013.pdf (visited on 09/12/2018).

[52] C. Matyszczyk. *Samsung's warning: Our Smart TVs record your living room chatter*. 2015. URL: https://www.cnet.com/news/samsungs-warning-our-smart-tvs-record-your-living-room-chatter/ (visited on 02/03/2018).

[53] R. Merkle. "Protocols for public key cryptosystems." In: *1980 IEEE Symposium on Security and Privacy*. 1980, pp. 122–133. URL: http://ieeexplore.ieee.org/document/6233691/ (visited on 12/27/2017).

[54] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 12/27/2017).

[55] V. Pureswaran and P. Brody. *Device democracy: Saving the future of the Internet of Things*. Tech. rep. IBM Institute for Business Value, 2014. URL: https://www-935.ibm.com/services/us/gbs/thoughtleadership/internetofthings (visited on 12/22/2017).

[56] *Quorum*. URL: https://github.com/jpmorganchase/quorum (visited on 01/15/2018).

[57] T. Rabin. "A simplified approach to threshold and proactive RSA." In: *Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings*. Springer Berlin Heidelberg, 1998, pp. 89–104. ISBN: 978-3-540-68462-6. URL: https://doi.org/10.1007/BFb0055722 (visited on 12/23/2017).

[58] M. K. Reiter and K. P. Birman. "How to Securely Replicate Services." In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 986–1009. URL: http://doi.acm.org/10.1145/177492.177745 (visited on 12/23/2017).

[59] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Internet Engineering Task Force, 2012. URL: https://tools.ietf.org/html/rfc6347 (visited on 09/23/2018).

[60] R. Roman, P. Najera, and J. Lopez. "Securing the Internet of Things." In: *IEEE Computer* 44.9 (2011), pp. 51–58. URL: http://ieeexplore.ieee.org/document/6017172/ (visited on 12/27/2017).

[61] *Samsung SmartThings*. URL: https://www.smartthings.com (visited on 02/03/2018).

[62] D. Schatsky and E. Piscini. *Deloitte blockchain survey 2017*. Tech. rep. Delloite, 2017. URL: https://www2.deloitte.com/us/en/pages/about-deloitte/articles/innovation-blockchain-survey.html (visited on 12/27/2017).

[63] M. Scherer. "Performance and Scalability of Blockchain Networks and Smart Contracts." Master's thesis. Umeå University, 2017. URL: https://umu.diva-portal.org/smash/get/diva2:1111497/FULLTEXT01.pdf (visited on 09/09/2018).

[64] H. Shafagh et al. "Towards Blockchain-based Auditable Storage and Sharing of IoT Data." In: *Proceedings of the 2017 on Cloud Computing Security Workshop*. CCSW '17. ACM, 2017, pp. 45–50. ISBN: 978-1-4503-5204-8. URL: http://doi.acm.org/10.1145/3140649.3140656 (visited on 02/05/2018).

[65] A. Shamir. "How to Share a Secret." In: *Commun. ACM* 22.11 (1979), pp. 612–613. URL: http://doi.acm.org/10.1145/359168.359176 (visited on 12/23/2017).

[66] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. Internet Engineering Task Force, 2014. URL: https://tools.ietf.org/html/rfc7252 (visited on 08/15/2018).

[67] V. Shoup. "Practical Threshold Signatures." In: *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'00. Springer-Verlag, 2000, pp. 207–220. ISBN: 3-540-67517-5. URL: https://www.iacr.org/archive/eurocrypt2000/1807/18070209-new.pdf (visited on 09/09/2018).

[68] W. Stallings. *The SET Standard & E-Commerce*. 2000. URL: http://www.drdobbs.com/the-set-standard-e-commerce/184404309 (visited on 07/30/2018).

[69] C. Stathakopoulou and C. Cachin. *Threshold Signatures for Blockchain Systems*. Tech. rep. Swiss Federal Institute of Technology, IBM Research, 2017. URL: https://cachin.com/cc/papers/hlftsig.pdf (visited on 09/23/2018).

[70] N. Szabo. "Formalizing and Securing Relationships on Public Networks." In: *First Monday* 2.9 (September 1997). URL: http://firstmonday.org/ojs/index.php/fm/article/view/548 (visited on 12/27/2017).

[71] *Tendermint*. URL: https://github.com/tendermint/tendermint (visited on 12/27/2017).

[72] P. Thakkar, S. Nathan, and B. Vishwanathan. "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform." In: *CoRR* abs/1805.11390 (2018). URL: http://arxiv.org/abs/1805.11390 (visited on 09/09/2018).

[73] *Threshsig*. URL: https://github.com/sweis/threshsig (visited on 07/23/2018).

[74] M. Vukolić. "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication." In: *iNetSeC*. Vol. 9591. Lecture Notes in Computer Science. Springer, 2015, pp. 112–125. ISBN: 978-3-319-39027-7. URL: http://vukolic.com/iNetSec_2015.pdf (visited on 12/27/2017).

[75] L. Welch and E. Berlekamp. *Error Correction for Algebraic Block Codes*. US Patent 4,633,470. 1986. URL: https://www.google.com/patents/US4633470 (visited on 01/07/2018).

[76] B. Wire, ed. *Datacenter Investments Critical to Internet of Things Expansion, IDC Says*. URL: http://www.businesswire.com/news/home/20150427005027/en/Datacenter-\%20Investments-Critical-Internet-Expansion-IDC (visited on 02/03/2018).

[77] B. Wire, ed. *Change Healthcare Introduces Enterprise Blockchain for Healthcare*. 2017. URL: http://www.businesswire.com/news/home/20170925005820/en (visited on 12/27/2017).

[78] L. Xu et al. "Enabling the Sharing Economy: Privacy Respecting Contract based on Public Blockchain." In: *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. BCC '17. ACM, 2017, pp. 15–21. ISBN: 978-1-4503-4974-1. URL: https://dl.acm.org/citation.cfm?id=3055527 (visited on 12/27/2017).

[79] Z. Zheng et al. "Blockchain Challenges and Opportunities: A Survey." In: *International Journal of Web and Grid Services* (December 2017).

[80] L. Zhou, F. B. Schneider, and R. V. Renesse. "COCA: A Secure Distributed Online Certification Authority." In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 329–368. URL: http://doi.acm.org/10.1145/571637.571638 (visited on 12/23/2017).